Universitetet i Oslo
Institutt for informatikk

Adresse-generator for dataflybaserte beregninger

Cand. Scient. Rapport

Kjetil E. Vistnes

November 2004

Abstract

Denne rapporten beskriver et design og en implementering av en adressegenerator for dataflytbasert beregning. Den kan generere adresser fra en 1, 2 eller 3-dimensjonal avbildning fra data som er lagret lineært i minnet. Beregningskretsen vil få data i en kontinuerlig strøm uten hull. Hver algoritme er spesifisert av et sett med parametere som lastes inn i FIFO-køer i bakgrunnen og byttes i løpet av en klokkepuls. Adressegeneratoren er programmert i VHDL, simulert og syntetisert. Implementasjonen har blitt utført på en Xilinx Virtex-II pro FPGA, og hardwaretestingen er gjort på et utviklingskort i fra Memec Design. Resultater i fra prosessen viser en hastighet på 144 MHz ved en generering 36 bit adresser. Videre foreslås metoder for å utvide fleksibiliteten. En løsning er å bruke en del av kretsen til å lagre forskjellige algoritmer slik at en delvis rekonfigurasjon kan gjøres like raskt som å bytte parametere.

Innhold

Figurer

Kapittel 1

Innledning

Innen områder som bildebehandling, signalbehandling, div. digitale filtreringer, videostrømoperasjoner og geometrisk modellering finnes det i dag en rekke løsninger og metoder for å beregne og behandle data. Det som ofte er karakteristisk for slike operasjoner er at de virker på en stor mengde data, der samme operasjon gjentas, og de er veldig krevende. Mange av applikasjonene nevnt over har strukturerte data. Det vil si at de ligger i blokker eller på en regulær måte i minnet som utnyttes. En del algoritmer for vitenskapelige beregninger benytter seg av array indekser i flere nøstede løkker, slik at sekvensen av data-adresser i programkjøringen viser et regulært mønster. Disse egenskapene kan utnyttes ved at applikasjonene utføres i en eller flere beregningsmoduler, som får inn data i en strøm og bearbeider disse etter en gitt funksjon eller operasjon. Denne måten å beregne data på kalles for dataflytbasert beregning (stream-based computation).

For å bruke en strøm av data, må dataene i strømmen være generert i riktig rekkefølge i henhold til hvordan beregningsfunksjonen vil ha dem. Typisk må data leveres hver klokkeperiode slik at en unngår tidsluker, der data uteblir. Genereringen av data-adresser gjøres av en adresse-generator. Temaet for hovedfagsoppgaven er å konstruere en adresse-generator for stream-based computation som er rekonfigurerbar for forskjellige funksjoner. Denne adresseringskretsen skal implementeres i FPGA-teknologi.

Kapittel 2

Dataflytbasert beregning

Dagens signalbehandlingsimplementasjoner er ofte DSP (Digital Signal Processor) eller FPGA baserte [5]. De fleste applikasjoner bruker von Neumann modellen, der prossesoren blir styrt av en instruksjonsstrøm. Denne modellen gjør seg ikke nytte av regulariteten i dataene, og parallellisering av operasjoner er ofte vanskelig fordi instruksjonene er avhengige av hverandre. Å bryte med von Neumann modellen gjør at man kan gjøre flere beregninger og øke båndbredden til minnet, siden ingen instruksjonshenting trengs [6]. det blir heller ikke lagret instruksjoner i minnet, noe som tar opp plass. Hovedforskjellen mellom en datasekvensmaskin og en von Neumann maskin er at systemet er kontrollert av en datastrøm istedet for en instruksjonsstrøm.

2.1 Dataflytbasert beregning innen video

Et område hvor ”stream computation” kan vise seg praktisk nyttig er video koding/dekoding og signalprosessering. Disse operasjonene er veldig krevende og trenger mye beregningskraft. ASIC’er og CPU’er har for det meste blitt brukt til dette og har til en viss grad møtt behovet for regnekraft i dagens algoritmer. Algoritmene i framtida vil kreve atskillig mer [19]. For digital video-data har typiske prosesseringssalgoritmer følgende egenskaper:

- I upakket form er informasjonen lagret i kompakte multidimensjonale arrayer

- Informasjonen er typisk ikke fast langs en dimensjon (tid)
- Det er enorme mengder data med i bildet

Typiske prosesseringssalgoritmer behøver ikke å aksessere alle dataene samtidig:

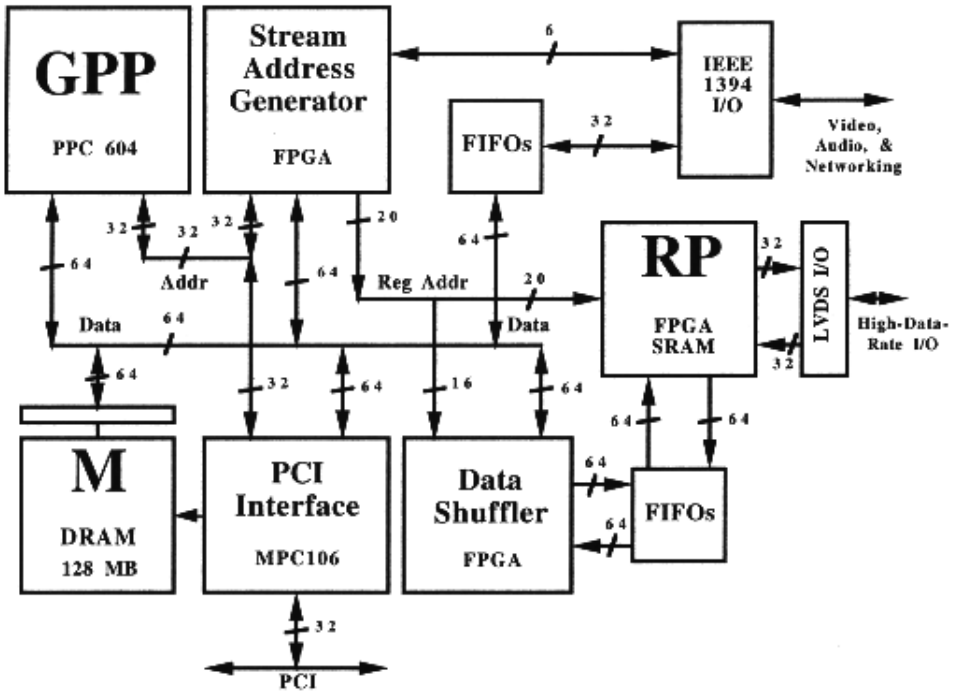
- Algoritmene opererer alene på et bestemt område av arrayen, med begrenset dimensjon, f.eks i en hel ramme eller deler av den
- Når det jobbes på små områder, så er data-aksesseringsmønsteret fast (bestemt)

Den grunnleggende ideen bak datastrømbaserte beregninger i video er at hver funksjon ikke blir sett på som å hente og lagre data i multidimensjonale arrayer (slik som upakket video-data er lagret). I stedet vil det si at en eller flere funksjoner jobber på en eller flere strømmer av data. Strømmen(e) er et resultat av en traversering av en avbildning (parametersett) som gjentas i flere iterasjoner ettersom datasettet blir oppdatert i løpet av tiden.

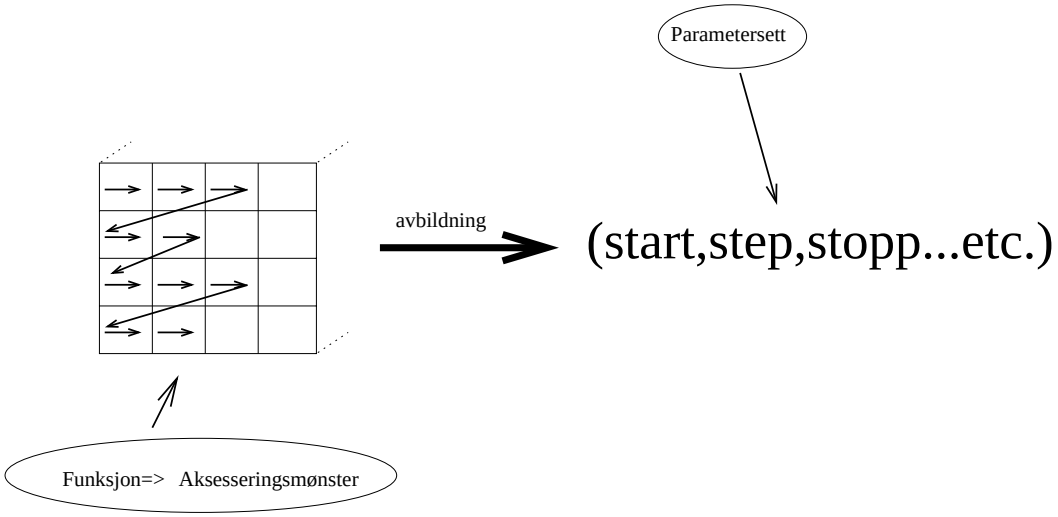
Chidi er et PCI-buskort for PC'er eller arbeidsstasjoner, som er utviklet ved MIT (Massachusetts Institute of Technlogy Media Laboratory) [16]. Det er laget for signalbehandling på strømmer av data, figur 2.1. Beregningene skjer på en FPGA (Altera 10k100, merket RP) sammen med en PowerPC 604e (merket GPP). Adresse generering (SAG) og buffer-logikk er også implementert på FPGA'er som er tilkoblet PCI bussen med sin kjappe forbindelse til RAM. Dette gir en datastrøm og en konvertering mellom minnets 64-bits ord og data-typene til mediet. Bufferingen blir gjort i Data Shuffler. Minne-kontroll blir gjort av en Motorola MPC106 PCI brikke, PCI Interface.

2.1.1 Et historisk tilbakeblikk

Begrepet strøm er blitt brukt ofte innen programeringssemantikk og programmeringsspråk siden 60'tallet. Motivasjonen for å bruke strømmer har variert, og hadde sin forløper i "coroutines", som er et sett med samarbeidende og samtidige operasjoner. Videre ble det utviklet programmeringsspråk som utnytter parallelliteten i algoritmer. Dette utviklet strømkonseptet videre [17][18]. Variable som endrer seg over tid, formelt representert av en strøm, ble brukt i forskjellige deklarative språk



Figur 2.1: PCI-bus kortet Chidi [16]



Figur 2.2: Prinsippet for stream computation i Chidi

slik som DFPL, VAL og Lucid. Strømmene i disse språkene var typisk en ordnet sekvens av skalarverdier, men også objekter var støttet. Disse strømmene var kun en-dimensjonale.

2.1.2 Strøm-generering i Chidi

Av interesse er strøm-genereringen (mekanismen) i Chidi, hovedprinsippet bak adresse-generatoren, som er en avbildning av en multidimensjonal data-array til en en-dimensjonal sekvens av data ved hjelp av et aksesseringsmønster. Et aksesseringsmønster er en parametrisert beskrivelse av en avbildning fra en fler-dimensjonal array til en en-dimensjonal sekvens. Dette mønsteret er et parametersett som definerer blant annet antall dimensjoner til arrayen, start, stopp og steg-verdier, for hver dimensjon, figur 2.2. Antallet avbildninger og mønstre fra et multidimensjonalt data-rom er ikke endelig og er vanskelig å bestemme på forhånd. Hver dimensjon kan være konstant, avgrenset av en parameter, eller være uendelig lang. Derfor kan adresse-generatoren i Chidi kun støtte visse mønstre som er vanlige for bildebehandlingsalgoritmer.

Aksesseringsmønsteret brukes til å beskrive hvordan multidimensjonale data-arrayer er lagret i lineære adresserom, som i vanlige minnebrikker. En annen måte å bruke det på, er at det beskriver hvordan en funksjon traverserer en multi-dimensjonal array, dersom det brukes direkte. En gjennomkjøring av en funksjons aksesseringsmønster gir et

sett med data. En strøm vil da vanligvis bestå av et aksesseringsmønster som er gjentatt over en strømkilde mange ganger, dvs. flere datasett kontinuerlig, følgende etter hverandre i strømmen.

En del aksesseringsmønstre er ikke data-avhengige. Dette gjelder algoritmer som f.eks. zig-zag scan og serpentine scan. I tillegg er det data-avhengige aksesseringsmønstre slik som motion estimation, entropi koding (kompresjons metode) og image warping (forandring ved forvrengning av bilder). Strømmer fra slike mønstre er ikke like lokale (i nærhet i minnet) og regulære som data-uavhengige mønstre, men det vil generelt være noe regularitet, til bruk av et cache-system [19]. Mange funksjoner produserer og prosesserer data i forskjellig hastighet og gjør at det trengs buffring før strømmen.

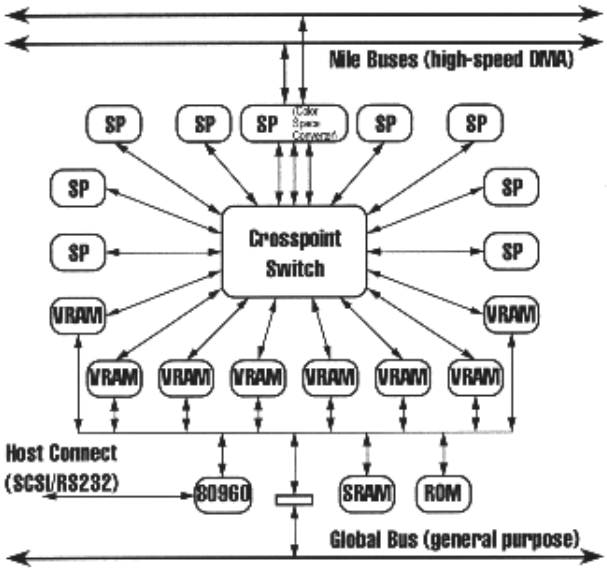
2.1.3 Parallellitet

En oppdeling av en algoritme i flere deler, og ved å la hver del bli prosessert i parallell, vil øke hastigheten til et system: flere beregninger pr. mikro-sek. Systemet kan ha flere prosessorer og også flere adressegeneratorer, slik som i *Cheops*, figur 2.3, der strømmene går i parallell til hver prosessor (merket SP). Det er 2 former for parallellitet: *kontroll-parallellitet* og *data-parallellitet*. *Kontroll-parallellitet* betyr at en algoritme deles opp i segmenter og hvert segment blir eksekvert samtidig av flere prosessorer. Hver prosessor behandler sitt ene segment av algoritmen. *Data-parallellitet* innebærer at hver av segmentene i algoritmen blir delt opp igjen og subdelene fra det oppdelte segmentet blir fordelt på hver prosessor og eksekvert i parallell. Strømmekanismen som er beskrevet, gjør nytte av begge typer.

Parallellitet gjør at en algoritme trenger mindre midlertidig lagring, siden en del operatorer slipper å vente på resultatet fra en eller flere foregående operasjoner. En produsent av en strøm og en konsument kan eksekvere i parallell, slik at algoritmens behov for midlertidig lagring blir sterkt redusert og buffring mellom hver ende blir mindre.

2.2 Andre beregningsorienterte maskiner

I mange sanntidsapplikasjoner (f.eks FIR filtrering) trengs det mange beregninger, og til en lav pris. Til og med avanserte prosessorer vil ofte

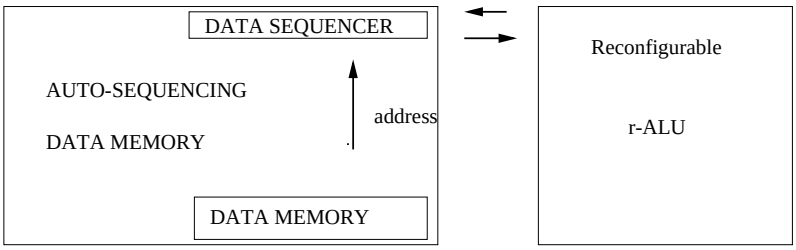


Figur 2.3: Cheops-prosessoren laget ved MIT [19]

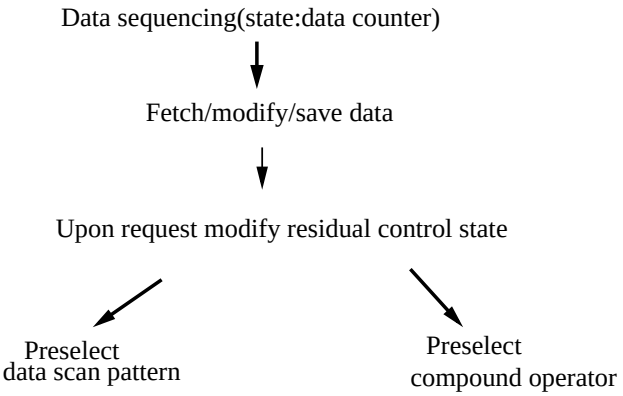
være trege og/eller for dyre. Også parallelle maskiner møter ikke kravene og er ofte veldig dyre [7]. Ved Universiteret i Kaiserslautern brøt de med von Neumann prosessering og introduserte en ny dataflyt paradigme, xputer [7]. Nøkkelord for denne maskinen er data-sekvensering, adresse-generering, rekonfigurerbar ALU og MoM (MapOriented Machine) [6]. En nærmere beskrivelse av Xputer og dens bruksområder kan ses i [7][8][6][9]. En vanlig alu i en CPU kan bare utføre en operasjon pr. tidsenhet. Xputer bruker flere alu'er implementert i FPAA'er (Field Programable Alu Array) som er rekonfigurerbare. Den blir drevet av en datasekvenser, figur 2.4, som henter, modifiserer eller lagrer data. Datastrømmen styrer flere alu'er, figur 2.5. Datasekvenseren inneholder flere adressegeneratorer som hver har et "scan-window" som jobber på et 2-D adresseplan. Disse generer adresser som brukes til å hente data til aluene.

Dataflyten i en von Neumann maskin er kontrollert av operasjonene i CPU'en. Det vil si at det er instruksjonene i programmet som bestemmer dataflyten til CPU'en, og disse instruksjonene følger i en instruksjonsstrøm. figur 2.6. Instruksjonene blir hentet fra minnet og utført etter hverandre. Mange instruksjoner må ofte hente data fra minnet, vente på data, eller lagres temporært i minnet. Programmet, som ligger i minnet, styrer instruksjonssekvensen, figur 2.7.

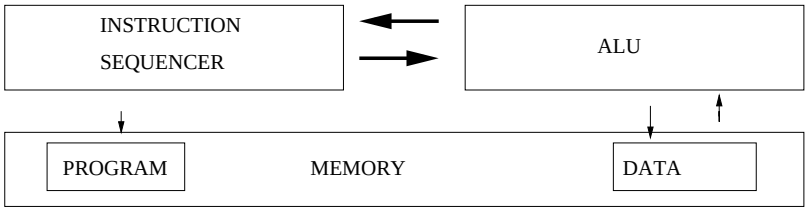
Ved Universitetet i Chemnitz [5] er det laget en hybrid



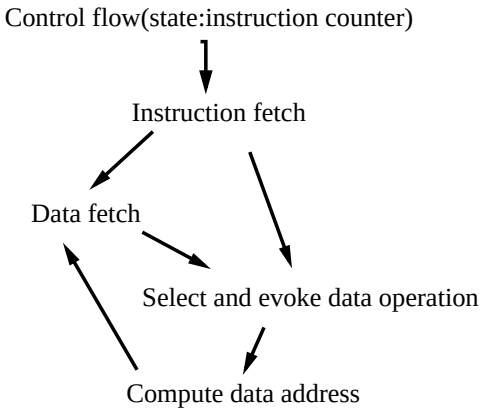
Figur 2.4: Datasekvens i Xputer [24]



Figur 2.5: Datastrøm i Xputer [24]



Figur 2.6: Datasekvens i Von Neumann prosessor[24]



Figur 2.7: Datastrøm i Von Neumann prosessor[24]

von Neumann/dataflytmaskin brukt til DSP. Ideen er at man kombinerer bruken av von Neumann og dataflytmodellen i en maskin, der ressurser som registerfiler, kommunikasjonsnettverk og eksekveringsenheter deles. Man kombinerer slik, fordi tilstandsmaskiner og deler av kontrollkode med avhengighet ikke egner seg til dataflytmaskiner. I hybridmaskinen vil de fleste operasjoner utføres på von Neumann måten, men dersom det er fordelaktig å bruke dataflytmetoden, vil denne metoden brukes for å behandle en mengde data. Maskinen er satt opp til å hente instruksjoner fra registeret før den går inn i dataflytmodus. Så vil en enkelt instruksjon konfigurere datastien, maskinen går i dataflyt modus og behandler data gjennom den konfigurerte datastien. Maskinen forlater denne modusen når en termineringsoperasjon slår til (f.eks en datateller slår ut) og fortsetter med von Neumann modus.

2.2.1 Fordeler og ulemper med dataflytmodellen sammenlignet med von Neumann-modellen

Fordelene med dataflyt er at man slipper kontrollflyt-overhead. Det er veldig mange kontrolloperasjoner i form av instruksjoner i vanlige maskiner. For hver datamanipulasjon trengs minst en kontrollhandling (instruksjon), og hver instruksjon må lagres i minnet. I tillegg slipper man adresseringsoverhead [6]. Det vil si at man slipper å bruke ekstra instruksjoner og dataoperasjoner for å beregne adresser, som det er mange av i von Neumann maskiner. Ofte brukes dataflytparadigme til vitenskapelige beregninger og beregningskrevende operasjoner. Disse er ofte ikke statiske, men dynamiske. Dette vil si at funksjonene og parametrene endrer seg ofte etter som vitenskapen, teknologien og metodene utvikles i tid. Dette gjør at rekonfigurerbarhet er viktig og dataflytmodellen kan lett implementeres i rekonfigurerbar teknologi, f.eks FPGA, som utvikles til å bli bedre og bedre. En FPGA-basert beregningsmaskin kan være best på applikasjoner med en stor grad av parallellitet, noe som en CPU har liten mulighet for [30] og i beregninger der store mengder av data skal behandles på en veldig lik måte.

Noen av ulempene med dataflytbaserte maskiner er at de er forbedringsfiendtlige. Operasjoner kan ikke optimeres i kompilering. Irregulær stukturert kode er nesten umulig å implementere. Mange nye typer av flaskehalser har oppstått. Det kan bli flere dataaksesseringskonflikter. Hvis man har flere adressegeneratorer som kanskje genererer samme adresse, kan dette føre til konflikter. Dataflytmodellen er også bare best på algoritmer med en stor del av parallellisering og en svært regulær

struktur. Mikroprosessorer er overlegne når det gjelder kompleks kontroll flyt og irregulære beregninger.

Kapittel 3

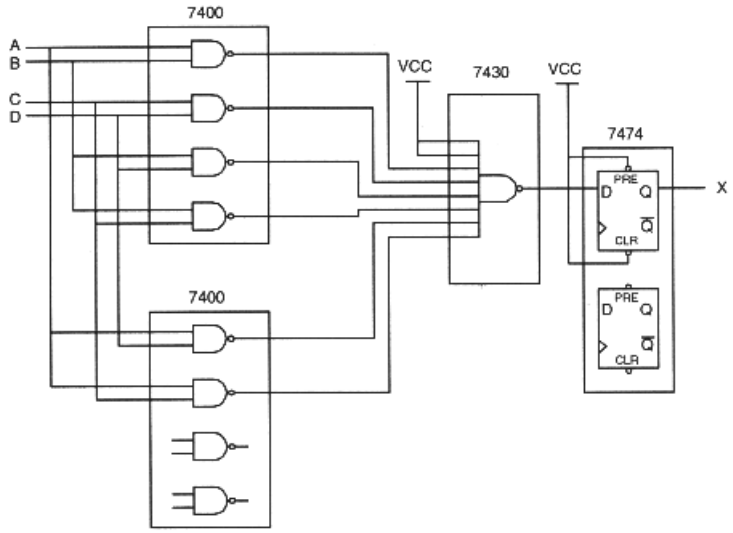
Programmerbar logikk

3.1 Noen av dagens vanligste hardware-løsninger

Det finnes mange valg en hardware designer må ta for å finne den rette balansen mellom hurtighet og generalitet når et design skal lages. Den mest vanlige løsningen er ofte å velge en tradisjonell microprosessor, som f.eks en Intel Pentium, eller en fra AMD. Disse brukes til vanlig i pc'er, og prosessorene er generelle. Det vil si at de kan utføre en hvilken som helst matematisk eller logisk operasjon. For spesielle og spesifikke operasjoner er de ikke designet spesielt, men de kan utføre disse funksjonene allikevel (spill som Quake3 og Word), ofte med hjelp av en co-prosessor (matte prosessor) eller en egen brikke på et 3-d skjermkort.

En annen mulighet er å få produsert en ASIC til oppgaven. Denne kretsen vil utføre nøyaktig den funksjonen som designeren vil og er spesielt finjustert slik at den er billigere å produsere (dog kan utviklingen av en ASIC være kostbar), kjappere og bruker mindre strøm. En veldefinert ASIC vil løse oppgaven som den ble designet for, men en liten forandring på problemet etter at ASIC'en er produsert, kan gjøre at den gamle ASIC'en må skrotes, og at en ny må lages.

På midten av 1980'tallet ble det introdusert en ny teknologi for å lage digital logikk: FPGA. Programmerbare kretser som fantes fra før var MPGA'er (small, slow mask programmable gate arrays) eller "programmable logic devices", PLD'er [15], som var store og dyre. FPGA'er er konfigurerbare. Funksjonen blir programmert ned på FPGA'en og lagret der, og kan rekonfigureres senere på forskjellige måter. Hvilke typer rekonfigurering

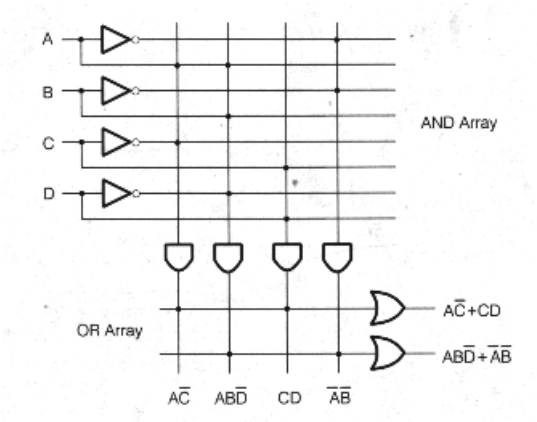


Figur 3.1: Nettverks forsterker implementert i TTL-logic [20]

som fins vil bli videre belyst i kapitelet.

3.2 Tidlige programmerbare kretser

Før utviklingen og bruken av PLD'er og FPGA'er var kommet ordentlig i gang, var Texas Instument sine TTL serier 54/74 av logiske kretser de mest brukte for å implementere logikk for multiplexing, dekoding, enkoding eller tilstandsmaskiner [20]. Kretsene fra TI var SSI (Small Scale Integrated) og MMI (Medium Scale Integrated), og inneholdt logiske porter, registere, tellere, shift-registere og aritmetiske deler. Et design med TTL logikk kan bestå av flere kretser fra 54/74 serien koblet sammen. F.eks. en "nettwork repeater", med 4 innganger og en utgang, kan lages som i figur 3.1. TTL kretsene inneholder kun NAND og NOR porter. Dersom man har f.eks AND og OR porter i designet sitt, må disse portene konverteres til NAND og NOR. Konverteringen kan f.eks. gjøres med DeMorgans teorem.



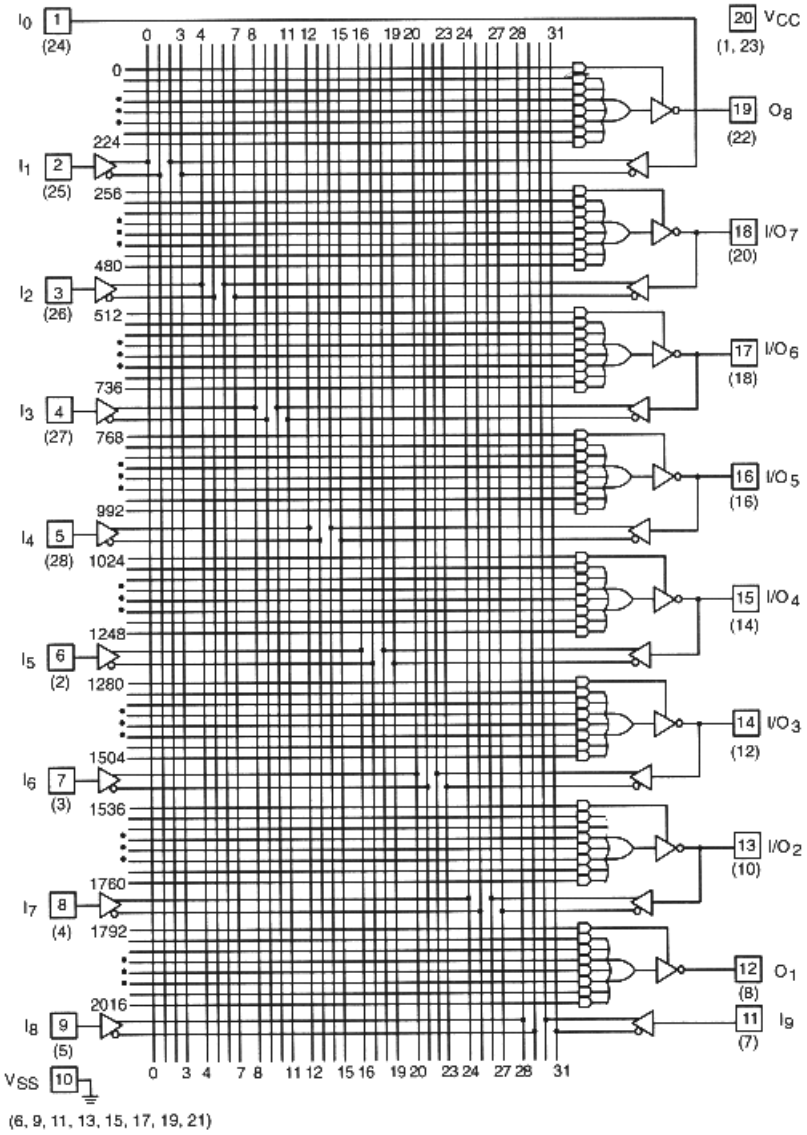
Figur 3.2: PLA-logikk [21]

3.3 Hva er en CPLD?

De enkleste av dagens programmerbare logiske kretser er PAL'er (Programmable Array logic). Disse kretsene ble utviklet på 70'tallet [21]. Som det går ut i fra navnet, så består an PAL av en array av AND porter, som er programmerbare, og en array av OR porter. Programmeringen blir gjort med EPROM, EEPROM eller FLASH teknologi.

Forgjengeren til PAL er PLA (Programmable Logic Array). PAL'en utviklet seg fra PLA'en på 70'tallet og en PLA består av et stort antall med AND og OR porter som er koblet sammen i en array. Alle innganger og deres komplementer blir ledet først inn til en AND array, figur 3.2. De vertikale utgangene fra AND portene ledes inn i en OR-array med OR-porter og utgangene der vil bli produkttermer.

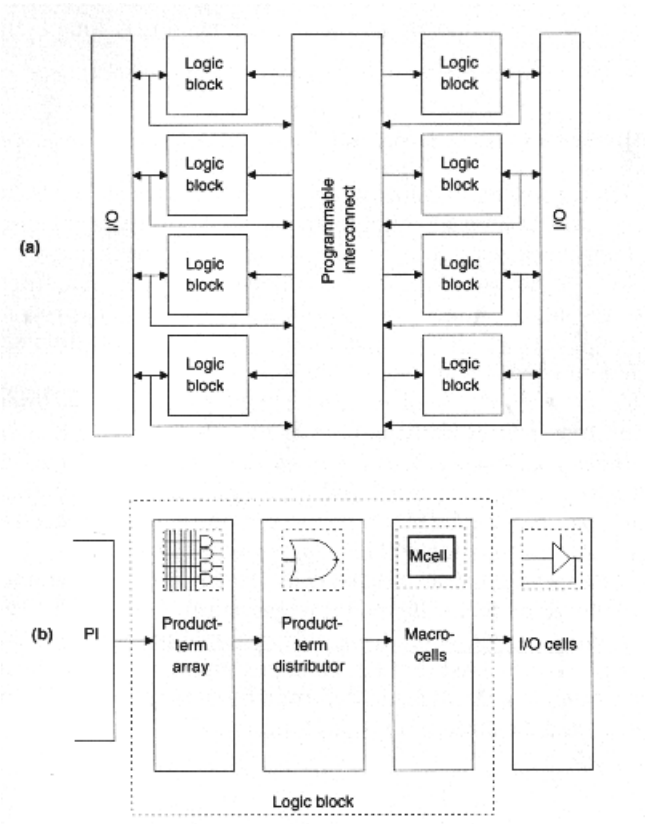
PAL 16L8 er en krets som er tilgjengelig fra flere fabikanter: AMD, Cypress Semiconductors og TI for å nebne noen. 16L8 har 16 innganger til AND arrayen og 8 utganger, figur 3.3. Den programmerbare AND arrayen består av 64 AND-porter og hver kan delta i en produktterm med hver av de 16 inngangene. O- arrayen er ferdig og hver av portene summerer 8 produktermer, der den åttende trigger et tristate buffer etter OR porten. 16R8 har registere med tilbakekobling etter hver produktterm, slik at man kan implementere tilstandsmaskiner, tellere, shift-registers etc. Når registere blir lagt til en PAL eller PLA, så blir den kalt en PLD (Programmable Logic Device) eller SPLD (Simple Programmable Logic Device).



Figur 3.3: PAL kretsen 16L8 [20]

Av enkle PLD'er som brukes i dag i industrien er 16V8, 20G10, 20RA10 og ikke minst 22V10 som var et lite gjennombrudd, fordi den inneholdt makroceller og variabel produkttermdistribusjon [21]. Det vil si at antallet produkttermer til hver OR port varierer mellom 8 og 16. Topp og bunn makrocellene allokerer 8 produkttermer, de midtre 16 og de andre 10, 12 eller 14, avhengig av hvor makrocellen er.

I stedet for å utvide en PLD med flere innganger, produkttermer og makroceller, så tok man og kombinerte flere PLD'er på en krets, slik at man fikk det som kalles CPLD (Complex Programmable Logic Device). En CPLD består av flere logiske blokker, og er en liten PLD, f.eks en 22V10. Hver logisk blokk har en produktterm array, produkttermdistrubusjon, som er ruting av utgangene til produkttermen, og makroceller 3.4a. De logiske blokkene kommuniserer med hverandre gjennom PI (Programmable Interconect) som ruter mellom I/O til blokker eller mellom blokker, figur 3.4b.



Figur 3.4: Oppbygningen til en typisk CPLD [20]

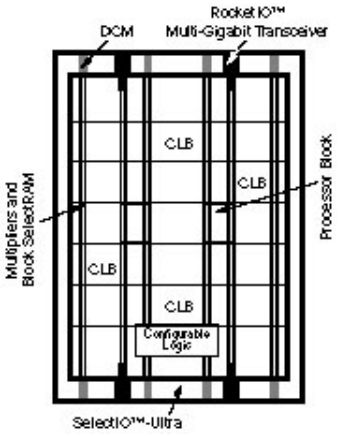
3.4 Hva er en FPGA?

FPGA'er blir programmert etter fabrikasjon og kan realisere hvilken som helst operasjon som passer inn i kretsens størrelse. Dette skiller en FPGA fra de fleste ASIC'er som normalt kun kan gjøre en eller få funksjoner. Dog finnes det også ASIC'er som kan lages programmerbare. Dersom logikken på FPGA'en er laget kun med kombinatorisk logikk, kan man si at en FPGA kan bare gjøre en "instruksjon" pr. klokkecykel. Denne instruksjonen sier hvordan kretsen er programmert og blir gjentatt for hver klokkecykel, så sant ikke kretsen blir rekonfigurert. Disse "store" instruksjonene blir delt opp i flere små primitive operatorer, som blir koblet til hverandre med ledninger.

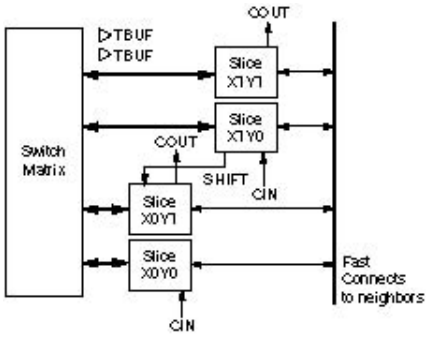
Man kan også programmere ned på en FPGA sekvensiell logikk med registre og/eller flip-flopper, slik at en funksjon blir delt opp i flere steg (instruksjoner som utføres etter hverandre). Dette blir mer sammensatte operasjoner, i motsetning til kun kombinatorisk logikk, og operasjonene består som oftest av flere klokkeperioder. I tradisjonelle prosessorer blir operasjoner lagret/lagd midlertidig og eksekvert sekvensielt i tid, med bruk av registre og minne til å lagre temporære resultater.

3.4.1 Oppbygningen til en typisk FPGA

Det er pr. dags dato flere teknologier til bruk for å utvikle en FPGA. FPGA'er er normalt konfigurert med SRAM, EPROM, EEPROM eller antifuse. Antifuse baserte FPGA'er kan bare bli programmert en gang, og brukes derfor ikke vanligvis i reprogrammerbare systemer, mens EPROM og EEPROM-baserte holder på konfigurasjonen etter at strømmen er slått av og kan bli elektrisk programmert etterpå. For å reprogrammere kretsen må det brukes høy spenning og er derfor ikke mulig mens kretsen er i bruk. EPROM og EEPROM har potensiale til å rekonfigureres, men de brukes heller ikke vanligvis til det [15]. En SRAM basert FPGA derimot kan bli programmert mens den er i bruk, men den må bli reprogrammert når maskinen (systemet) slås på. Vanligvis blir da konfigurasjonsbitene lagret i ROM i tilknytning til kretsen. På grunn av at SRAM baserte FPGA'er er reprogrammerbare mens kretsen er i bruk, blir implementasjonen av kretsen mer fleksibel. Siden de er lette å reprogrammere, kan konfigurasjonen lett forandres for å rette på feil eller for å oppgradere systemet. I dette avsnittet blir en SRAM-basert FPGA fra Xilinx brukt som eksempel, da en slik FPGA brukes som krets i oppgaven.



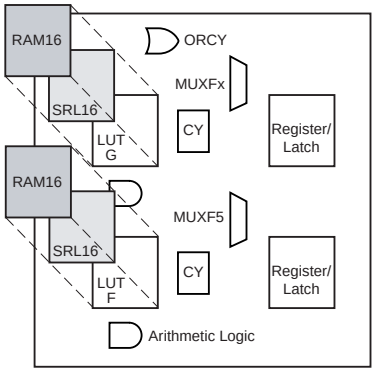
Figur 3.5: Xilinx Virtex-II Pro[23]



Figur 3.6: Xilinx Virtex-II Pro CLB[22]

En Virtex-II Pro FPGA består av et antall logiske blokker (CLB) som er forbundet med hverandre i et nettverk av forbindelseslinjer (rutingskanaler). Dette nettverket er realisert på FPGA'en som et array av bit-prosessereringsenheter som blir programmert etter fabrikasjon. Rutingslinjene er forbundet med I/O-blokker, som forbinder nettverket av CLB'er med I/O-pinner, figur 3.5. De FPGA'ene som er grovkornet, det vil si at det er få CLB'er, har kraftige programmerbare blokker. De med en finere struktur derimot har flere enkle blokker som prosesserer færre bit. En CLB på en Virtex-II Pro inneholder 4 slicer, figur 3.6.

Hver slice inneholder to funksjonsgeneratorer som hver har en inngang på 4 bit. Funksjonsgeneratorene kan bli programmert som en 4-inngangs Lookup table (LUT), 16 bit med distribuert SelectRAM+ minne eller som et 16-bit shift-register. I tillegg har hver slice registre, mux'er etc., figur 3.7. Hver av de to LUT'ene brukes til å implementere bolske funksjoner med 4 innganger. Multiplekseren MUXFx i slicen kan bruk-



Figur 3.7: Xilinx Virtex-II Pro Slice[22]

es til å kombinere Lut'er for funksjoner med 5 til 8 innganger. Spesielle funksjoner med 9 innganger kan implimenteres i en slice ved hjelp av MUXF5.

3.5 Praktisk bruk av FPGA

3.5.1 Hvorfor bruke FPGA-teknologi?

Dagens FPGA'er har gjennom mange års utvikling etterhvert fått et stort antall porter: Port-tettheten på kretsen øker for hvert år og dette gjør at FPGA'er nå kan utføre nyttige beregninger, spesielt brukt i matematiske og signalbehandlingsrelaterte beregninger, men også mer og mer brukt som akselerator i PC'er og arbeidsstasjoner. Visse oppgaver er spesielt egnet til å bli implementert i rekonfigurerbar hardware og mange multimedia-algoritmer har følgende særpreg [16]:

- ***Dataparallellitet*** med minimal avhengighet mellom minneenheter eller *regulære blokker med data*
- ***Kontinuerlig båndbredde*** inn og/eller ut av systemet
- ***Variabel lengde eller uvanlige datatyper*** som ikke er spesielt bra støttet av vanlige CPU'er
- ***Lange beregnende pipelines*** uten hopp

- ***Dataflytting (omstokking)*** som kan stresse en CPU's cache eller bus-interface

3.5.2 Områder hvor FPGA-teknologi brukes

Det har vist seg at en del applikasjoner, som signalprosessering, emulering, kryptering og videostrømoperasjoner, blir utført mye fortere på en FPGA enn en tradisjonell von Neumann prosessor. The splash system [1] har vist ytelse på genetisk streng sammenligning, som er nesten 200 ganger raskere enn på tallknusere. The DECPeRLe-1 [2] system har demonstrert stor ytelsesforbedring på mange appliksjoner, inkludert RSA kryptografi. En av de mest lovende områdene til bruk av FPGA-basert teknologi kan være ASIC emulering [15]. Formålet med emuleringen er at ASIC-utvikleren vil forvisse seg om, ved å teste ut om kretsen som er designet, virker korrekt i henhold til spesifikasjonene. Man kan også bruke en emulator i software, men det viser seg at disse utfører emuleringen veldig sakte. I logisk emulering kan ASIC-kretsen bli representert på et multi-FPGA system, som gir en flere gangers ytelsesforbedring enn software-simulering. Kretsen bruker logisk emulering dvs. at portnivåbeskrivelsen til kretsen blir programmert ned på multi-FPGA'en. Multi-FPGA systemet som blir brukt, er en prefabrikert, reprogrammerbar logisk maskin som kan bli konfigurert til å implementere den ønskede kretsen.

3.6 Rekonfigurering og context switching/self reconfiguration

De siste seriene av FPGA'er fra produsenter som Xilinx og Altera tilbyr raskere konfigureringstid enn de tidligste modellene, som trengte flere sekunder for å bli programmert. På en Xilinx Virtex kan en lese eller skriveoperasjon med 156 bytes (minste størrelse) konfigurasjonsminne gjøres med en programmeringstid på 3.12 μs. [11]. Dette har åpnet døra for det som kalles "configurable computing". Inntil de seneste årene har det vært for det meste software som kan oppgraderes, men med kjappere konfigurering kan også hardware oppgraderes hos brukeren. Oppgraderingen realiseres da som utskifting av konfigurasjonen. Hardwaren i et system kan dermed forandres under kjøring, avhengig av beregningsalgoritmen eller hvilke inndata som kommer inn i systemet og omgivelsene rundt [10].

Bruken av rekonfigurering og graden av denne kan grovt sett deles inn i 3 typer [11]:

- **Statisk:** Ingen rekonfigurering av systemet i løpet av dets levetid. Denne bruken er mest vanlig i dagens systemer.
- **Dynamisk:** Konfigurasjonen eller deler av den erstattes i blant for å forandre systemet. Denne typen bruk av rekonfigurerbarhet blir mer vanlig og er nyttig hvis det oppdages feil som må rettes eller funksjonaliteten til systemet skal forandres.
- **Context switching:** En oppgave kan deles opp i flere mindre deler, og imellom hver del byttes context. Det optimale er da å ha et sett av konfigurasjoner lagret internt på FPGA'en, context'er, som blir skiftet ut ettersom operasjonen kjøres og hver context switching kan ideelt sett ta kun en klokkecykel. Den totale tiden for en oppdelt oppgave blir da kortere, gitt at rekonfigureringstiden er så kort at det lønner seg å dele opp en oppgave, i forhold til det å basere seg på kun en original, grunnoperasjon. Deler av FPGA'en rekonfigurer seg selv mens en oppgave kjører på en annen del, eller hele funksjonaliteten byttes ut, "on the fly", og slike systemer blir ofte kalt selv-rekonfigurerbare (self-reconfigurable). Andre termer som også brukes er: *Dynamically Reconfigurable og Run-time Reconfigurable* [15].

Ettersom FPGA'er blir raskere å rekonfigurere og har flere logiske blokker på kretsen, vil context switching bli mer aktuelt, og få flere bruksområder. Hittil har rekonfigurerbar logikk, FPGA etc. blitt brukt innen områder som bildegjenkjenning, mønster-sammenligning (håndskriftsgjenkjenning og ansiktsidentifisering) og video-overføring [11]. I framtiden kan det også være aktuelt å lage mer fleksible maskiner enn nåværende CPU'er og ASIC'er.

Det er mulig å bruke rekonfigurering på kommersielle FPGA'er eller nye, egenutviklede FPGA lignende systemer, som det er gjort arbeid på [11]. Vanlige FPGA'er er da SRAM baserte, fordi de kan reprogrammeres raskt, uten å forandre på hardwaren eller rerute inne i kretsen. Dette gjør at de er best egnet til rekonfigurering, og er mer fleksible, i motsetning til vanlige gate arrays (CPLD, MPGA). Den raskeste måten vil være å ha context switching med et sett av konfigurasjoner lagret og bytte i løpet av en klokkecykel. NEC har utviklet en brikke som har disse egenskapene [13] og det blir framsatt et forslag i [14]. Det finnes pr 2002 ingen kommersielle tilgjengelige FPGA'er som tilbyr dette i en klokkecykel og

konfigurasjonenene må lastes ned eksternt. I tillegg er konfigurasjonstiden for lang [11]. En oppgave som nevnt over, som deles opp i flere deler og bruker context switching, bør deles opp i flere større deler, for å redusere overheaden med å bytte konfigurasjon. De fleste FPGA'er krever at hele konfigurasjonsstrømmen lastes ned i en operasjon, som gjør at tiden for å programmere FPGA'en øker proporsjonalt med størrelsen. Xilinx produserer serien Virtex som kan bli delvis rekonfigurert: Tiden for å bytte til en ny deloppgave blir kortere. På en Virtex 1000 tar et sett med konfigureringsdata (765 968 byte) ved 50 MHz 15,3 ms å laste ned.

Alle kommersielle FPGA'er laster ned konfigurasjonen utenfra og med de egenskaper som nevnt over, er det laget/framsatt forslag om egendefinerte FPGA'er som kan rekonfigurere seg selv "on the fly" (se Context switching). Disse egendefinerte FPGA'ene kan utføre mer sammensatte, komplekse "grunnoperasjoner" og inneholde spesielle koblingsnettverk. En fordel med context switching/dynamisk rekonfigurering som def. i [11], er at man sparer mye hardware, da man bruker de samme ressursene på kretsen. I stedet for at en stor funksjon tar opp mesteparten av logikken på en krets, kan den deles opp slik at deler av den kjøres etter hverandre på samme "del" av kretsen, den delen av FPGA'en der logikken byttes ut.

I [11] er det laget et nytt forslag for å implementere Context switching på en Virtex FPGA. Dette forslaget er basert på tidligere arbeid med å implementere en egendefinert FPGA på en kommersiell FPGA. Dette designet består av flere User Defined Configuration Registers (UDCR), hvor kun et er aktivt om gangen, og bytting mellom registrene er mulig i en eller få klokkecykler. Den UDCR'en som er aktiv konfigurerer den rekonfigurerbare hardwaren. Ulempen med dette er at en stor mengde logikk går med på å definere den egendefinerte FPGA'en, til ruting og UDCR'er. Hardware overheaden blir stor, fordi det blir lite logikk igjen til rekonfigurerbar logikk i forhold til mengden av logikk originalt tilgjengelig på kretsen. Konfigurasjonsbitstrømmen blir lastet ned under kjøring, når Context switching skal gjøres. Dette blir gjort på en pipeline-lignende måte, for å redusere tiden. FPGA'en er delt opp i et sett med sub-FPGA'er som veksler på å være aktive og blir konfigurert hver for seg. FPGA'en blir en pipelinet enhet hvor konfigurering er pipelinet med eksekvering.

3.7 Utviklingen av FPGA-teknologien. Dagens platform og mulig fremtidig bruk

Utviklingen innen FPGA-teknologien de siste årene gjør at disse kretsene har gått fra å være et nisjeprodukt innen programmerbare kretser, til å være det som blir brukt til vanlig. CPLD'er er dermed blitt utdatert [3]. Virtex-II Pro FPGA fra Xilinx er pr. dags dato en av de raskeste programmerbare krets med 125 000 logiske celler og millioner av porter [4]. Denne kretsen inneholder Rocket IO for høy båndbredde for overføring av data. I tillegg også opptil 4 IBM PowerPC 405 prosessor hver på 400 MHz [23]. Ytelsen og hastigheten til de beste kommersielle FPGA'er øker på grunn av konkurranse, ny teknologi og innovasjon.

Reprogrammerbare systemer har vist stor ytelesesforbedring innen mange typer applikasjoner. Teknologien som brukes for å oppnå gode resultater innen beregningskrevende operasjoner er enten ved bruk av vanlige, generelle, kommersielle FPGA'er [11][32][33][35] eller egne, spesial bygde system [9][13][14][26][27] som er rekonfigurerbare. Disse systemene kan ha en til to kretser eller systemer med flere hundretusen FPGA'er. I tillegg inneholder mange systemer minne, DSP'er, CPU'er etc.

Kommersielle FPGA'er innen strømbaserte beregninger blir brukt i Chidi [16] systemet som er et dataflytbasert system. De blir brukt både til adressegenerering og beregning. Adressgeneratoren er ikke run-time rekonfigurerbar.

Videre vil vi i framtiden se systemer der FPGA'er og CPU'er utfyller hverandre. Dette kan være generiske co-prosessorer laget i FPGA eller reprogrammerbare enheter inne i prosessoren.

En utfordring vil være å lage software og applikasjoner som kan "mappe" konfigurasjoner og algoritmer ned på rekonfigurerbare systemer kjapt og effektivt. Det blir gjort automatisk av verktøyet, slik at en sofware programmerer ikke trenger å kunne noe særlig hardware. I tillegg er mange reprogrammerbare system oppbygd av FPGA'er som er lagd for generell logikk. Disse kretsene er ikke optimalisert for rekonfigurering. Rekonfigureringstiden har hittil vært for lang, men det er laget system i de siste årene, nevnt i foregående delkapittel, som kan bytte konfigurasjon på en klokkecykel. Det kan ofte derfor være at det lønner seg å lage egenproduserte systemer i stedet.

Kapittel 4

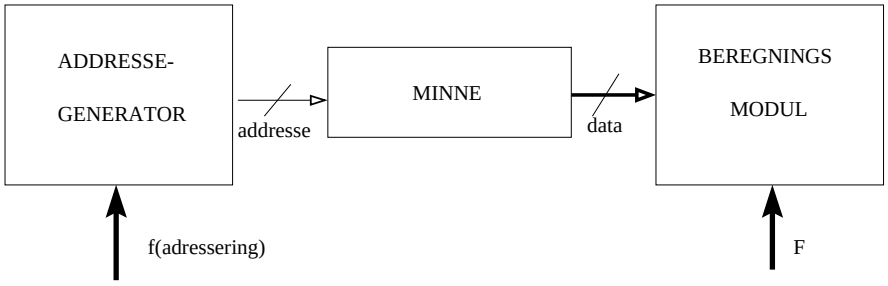
Implementasjonen av adresse-generatoren på en FPGA

Dette kapitlet beskriver i mer detalj hvordan adresse-generatoren er implementert i FPGA-teknologi. Her vil en overordnet beskrivelse inngå, spesielle føringer og krav som løsningen er bygd etter vil også bli beskrevet, samt en videre forklaring på implementasjonen i VHDL.

4.1 Adresseringskretsen i systemet

Systemet består av en adresseringskrets og en beregningskrets, figur 4.1. Begge disse modulene kan stå på samme PCI-kort eller på to forskjellige. Beregningsmodulen lages i FPGA, og er dermed rekonfigurerbar og fleksibel, avhengig av operasjonen F. Dataene blir så sendt fra minnet i en helt bestemt rekkefølge til beregningskverna, avhengig av hvordan denne er programmert, dvs. hvilken rekkefølge av dataene denne krever. Dataene i minnet må selvfølgelig oppdateres når funksjonene byttes eller de trengs ikke å byttes ut dersom neste funksjon bruker de samme dataene.

Hver funksjon (F) i beregningskretsen har sin respektive adresseringsfunksjon, f (adressering). I praksis er det et parametersett som blir lastet inn i adresse-generatoren, og som den genererer adresser etter. Minnebussen på et hovedkort består av to deler: Adressebussen og databussen. Utgangen fra adresse-generatoren vil gå til PCI-bussen og videre



Figur 4.1: Super-operasjon m/adr-gen.

på adressebussen til minnet. Adressebussen brukes til å velge hvilken minneadresse som data skal leses eller skrives til i minnet. Datumet fra minnet vil bli lest ut på databussen.

Et system som nevnt over har sine fordeler og ulemper som vil bli diskutert nærmere i kap 6. Det er hensiktsmessig å ta den analysen etter at resultater som hastighet og ressursbruk er presentert i neste kapitel. Det kan da være på sin plass å se på mulige andre løsninger.

4.1.1 Designspesifikasjoner

Vi har satt noen krav og føringer som lager en ramme for designet. En del av disse er fremsatt i oppgaveteksten og vi følger dem; adresseringskretsen skal virke med disse “parametrene”.

Adresse-generatoren skal levere et nytt datum for hver eneste klokkeperiode: den skal generere en ny adresse for hver klokkecykel. Det skal ikke være noe overhead ved ekstra klokkepulser innimellom dataene. Dette er veldig viktig fordi beregningskretsen ikke skal stå og vente på gyldig data. Beregningspotensialet til kretsen skal ikke være ubrukt. Målet er at tiden som algoritmen bruker er minst mulig. Å lage en adresseringskrets som gir et nytt datum for hver klokke puls gjør at beregningskretsen ikke trenger ekstra logikk for å takle No Operation (NOP). Beregningskretsen vil bruke mindre plass på en FPGA. Mindre resssurser til kontrollogikk gir mer ressurser til beregnende logikk på beregningskretsen.

Datastrømmen skal være feilfri. Dette innebærer at ingen data skal bli gjentatt eller at feil datum kommer innimellom gyldige. Dette er et absolutt krav. Den skal virke korrekt i henhold til en gitt algoritme og

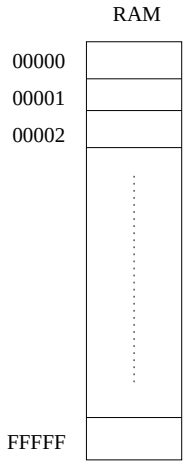
sitt parametersett som avgrenser hvilke adresser i adresserommet som skal brukes i genereringen av datastrømmen.

Siden beregningsmodulen er rekonfigurerbar, er det også naturlig at adresse-generatoren er det. Den skal implimenteres i rekonfigurerbar logikk, FPGA. Adresse-generatoren følger beregningsmodulens funksjon og rekonfigureres samtidig som funksjonen blir byttet. Dette vil si at adresse-generatoren skal bytte algoritme når beregningskretsen gjør det. Å bytte algoritme for hele systemet innebærer for adresseringskretsen at den skal fortsette å sende adresser ut i en strøm uten hull. Den skal ikke bare sende et nytt datum for hver klokkepuls i en gjennomkjøring av en algoritme, i praksis et parametertsett, men også når algoritmen byttes. Bytte av algoritmer skal være fleksibelt og ikke føre til noen hull i datastrømmen.

I tillegg til disse kravene som er gitt i oppgavebeskrivelsen, så er det også en del føringer i konstruksjonen av adresse-generatoren som må gis i form av egensatte krav. Hastigheten til adresse-generatoren bestemmer hastigheten til beregningskretsen. Med hastighet menes klokkefrekvensen i adresse-generatoren. På grunn av forsinkelser av signaler gjennom portene og rutingen i FPGA-kretsen, så vil klokka ha en maksimums frekvens som den kan kjøre på. Denne frekvensen bestemmer hvor fort hele systemet går. Hvor lang tid en algoritme bruker, er bestemt av hvor mange data den trenger og hvilken frekvens disse blir tilgjengelige for beregningsmodulen. Et mål bør være å få så høy klokkefrekvens som mulig. Ut fra de praktiske forholdene som gjelder for denne oppgaven, har jeg i utgangspunktet satt meg et mål på en klokkefrekvens på over 100 MHz.

De siste prosessorene ifra Intel Pentium serien, fra Intel Pentium II og utover, har en bredde på adressebussen på 36 bit [25]. Dette betyr at data kan ligge i et minnerom som er 36 bit stort. Derfor har jeg valgt at adresse generatoren støtter minnerom helt opp til 36 bit; utadressen er derfor 36 bit bred. I tillegg så passer det med bredden på fifo-modulen som er ferdiglaget fra Xilinx. Dersom hovedminnet har et adresserom som er større enn 36 bit, kan data som lagres der, bare adresseres i de nederste 36 bitene av minnet f.eks. Et diskret segment av minnet blir brukt tilsvarende adresserommet til adressegeneratoren.

Vi vil bruke Virtex-II pro fra Xilinx som “target device”. Dette er en FPGA med relativt mange I/O pinner og mange porter til logikk. Hvor mange inn og utganger til systemet som kan brukes, blir begrenset av tilgjengelige I/O utganger på kretsen. Antallet som systemet bruker vil ha betydning for hele systemet og vil bli diskutert mer i kapittel 6 sam-



Figur 4.2: Vanlig organisering av RAM

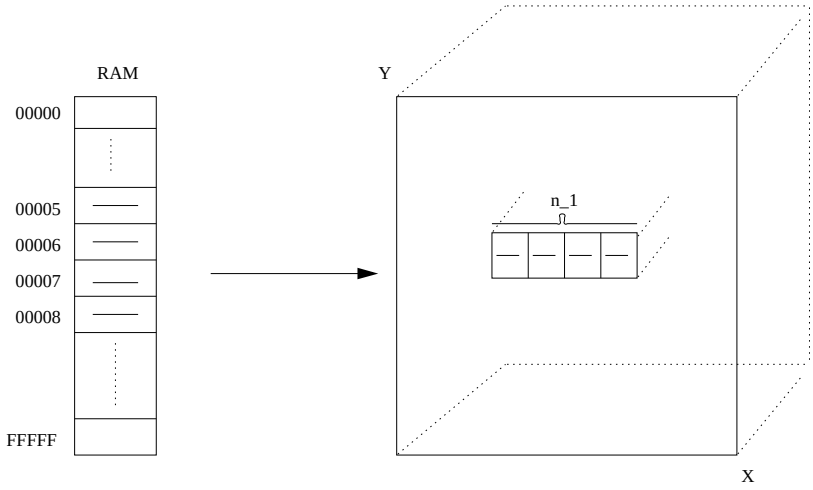
men med hvor mye plass adresse-generatoren opptar på FPGA-kretsen. Hastighet vil være det viktigste kontra plassbruk, men plassbruk kan ha noe å si for hvor mange adresse-generatorer man kan få plassert ned på en FPGA-krets. I motsetning til en CPLD, så kan man med en FPGA implementere mer komplisert logikk. Virtex-II pro inneholder også blokkram som brukes i adresse-generatoren. Derfor bruker vi ikke CPLD.

4.2 Dataavbildning for adresse-generatoren

Data er typisk lagret sekvensielt i det fysiske minnet og ligger i en stor lineær array som er adressert fra 00000 til FFFFF f.eks, avhengig av størrelsen til minnet, figur 4.2, kjent som RAM. Utfordringene blir da å overføre dette til et virtuelt datarom, avbildningsrom, som adresse-generatoren "jobber" etter. Data som man trenger i beregningen ligger i blokker i minnet med en fast avstand seg imellom. Dette kan utnyttes til å lage et datarom i 3 dimensjoner x, y og z, eller flere dimensjoner.

4.2.1 Avbildning i 1 dimensjon

La oss begynne med et enkelt eksempel der man bare vil ha en enkelt blokk fra minnet som data til beregningen, figur 4.3. Denne blokken har f.eks. adressene 00005 til 00008. Det første datum har adresse 00005 og blir base-adressen som man går etter. Base-adressen sendes til minnet,

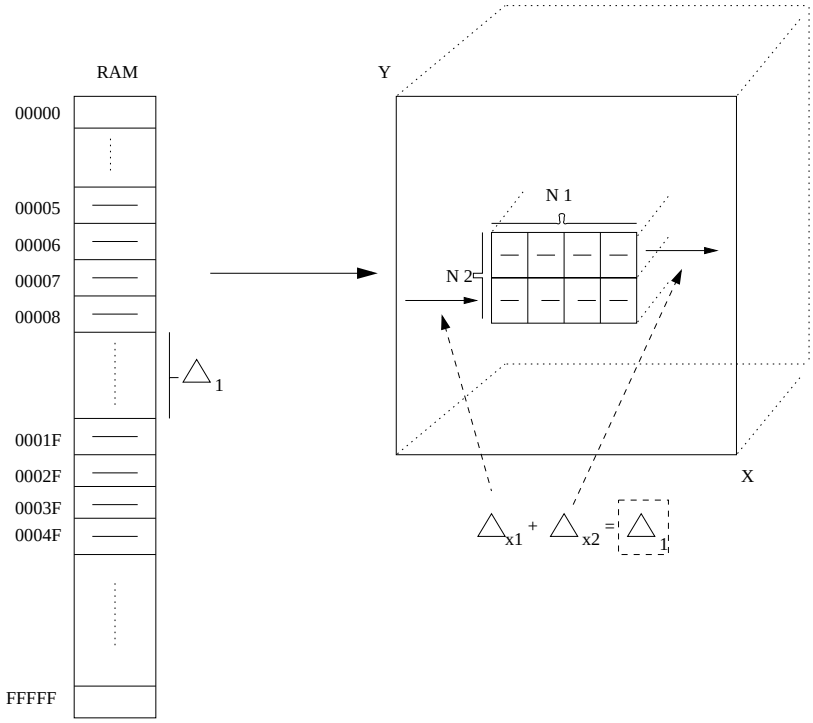


Figur 4.3: En minne blokk i avbildningsrommet

som sender denne minneadressen sitt tilsvarende data til beregningskverna. For å beregne neste adresse blir base-adressen inkrementert med 1. Slik går det helt til man har aksessert den siste adressen i minneblokken (00008). Antallet steg man teller, bestemmes av hvor stort arrayet er. La oss kalle denne parameteren for N1. Denne bestemmer hvor langt i avbildningsrommet man skal gå i x-retning.

4.2.2 Avbildning i 2 dimensjoner

En egenskap med mange algoritmer er at dataene er lagret regulært i minnet, figur 4.4. De ligger i blokker som er avgrenset med en gitt avstand imellom blokkene, et diskret mellomrom. Hvis dette mellomrommet er konstant og likt for alle blokkene, kan dette mellomrommet, Δ_1, benyttes. Hver blokk i minnet blir liggende i datarommet som et plan i det 3 dimensjonale rom, der hver blokk ligger under hverandre i xy-planet. I avbildningsrommet vil Δ_1 bli mellomrommet fra siste adresse i en blokk til den første i neste. Δ_1 angir hvor mange minneadresser det er fram til neste linje i avbildningsrommet. Antallet linjer i xy-planet adresse-generatoren skal traversere er N2.



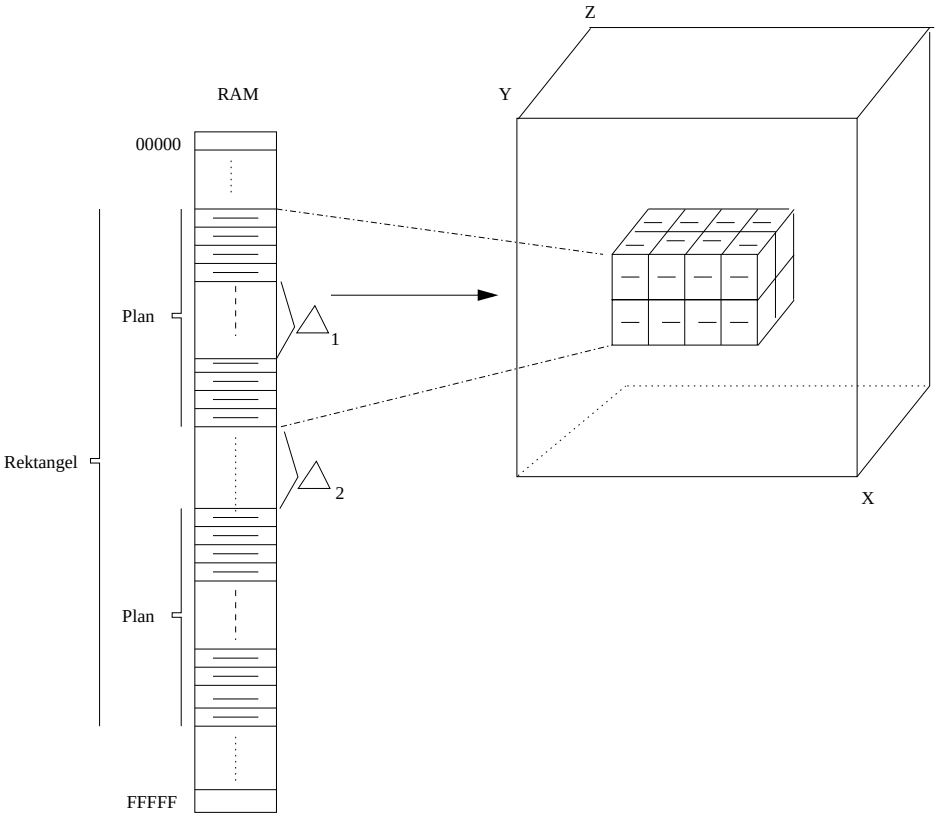
Figur 4.4: Plan i avbildningsrommet

4.2.3 Avbildning i 3 dimensjoner

Hvis flere slike områder i minnet som består av minneblokker, går igjen, kan man få dannet kuber eller rektangler i avbildningsrommet. Planene ligger parallelt ved siden av hverandre i z-retningen i rommet, figur 4.5. Avstanden mellom hvert plan, det vil si fra det siste minnepunktet i et plan til det første i det neste, er Δ_2. Når man går ifra aksessering av et plan til det neste, beveger man seg et steg i z-retning.

4.2.4 Avgrensning

Det som er beskrevet hittil er kun en avbildning i 3 dimensjoner fordi det er lett å visualisere. Dersom man har en avbildning i form av en kube eller et rektangel, som består av 2 eller flere plan, kan slike kuber ligge etter hverandre minnet, med lik avstand i mellom seg, og utgjøre en ny "subblokk" i det lineære minnet. Dette vil bli en fjerde dimensjon. En linje i et plan i avbildningsrommet blir en sub-subblokk i den blokken som utgjør et plan, og et plan blir en subblokk i en kube (rektangel). Slike re-



Figur 4.5: Rektangel i avbildningsrommet

gulære deler, subblokker, subarrayer som ligger i hverandre i et hierarki kan være stort og avbildningen vil derfor få et stort antall dimensjoner, avhengig av kompleksiteten til algoritmen og aksesseringsmønsteret .

Vi har valgt å lage adresse-generatoren slik at den kan generere adresser fra en avbildning som spenner over 1, 2 eller 3 dimensjoner i avbildningsrommet. De elementære blokkene i en avbildning, linjene, må være like store (lange), figur 4.4. Linjene kan heller ikke ha noen mellomrom i seg; de må være konsistente.

4.2.5 Parametersettet

Aksesseringsmønsteret er i praksis et sett med parametre. Dette parametersettet avgrenser og bestemmer hvilke adresser som skal genereres. Hver funksjon eller deler av en funksjon, har sitt eget aksesseringsmønster og parametersett, så et parametersett kan være en hel gjennomkjør-

ing av en prosess eller deler av den.

For å benytte seg av stream computation trenger ikke data som er lagret i minnet å være opprinnelig lagret med et regulært mønster. Mange algoritmer kan inneha en regulær natur i den forstand at de benytter seg av samme data i flere iterasjoner, og/eller at algoritmen er slik at den bruker blokker av data i minnet.

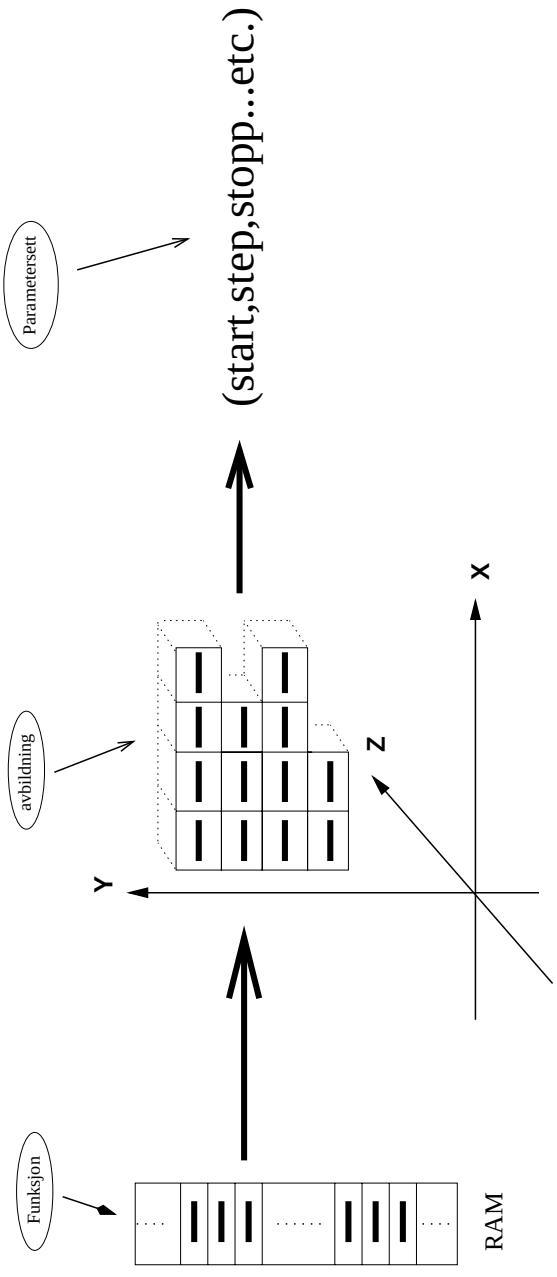
Uansett om det er bevisst at det er regulært lagret data i minnet som er grunnlaget for parametersettet, eller om funksjonen er "regulær", så vil adresse-generatoren bruke et parametersett som vist i figur 4.6.

For å generere en adressestrøm som er basert på en avbildning i maks 3 dimensjoner og hvor det er avgrensinger, trengs det kun følgende 6 parametre:

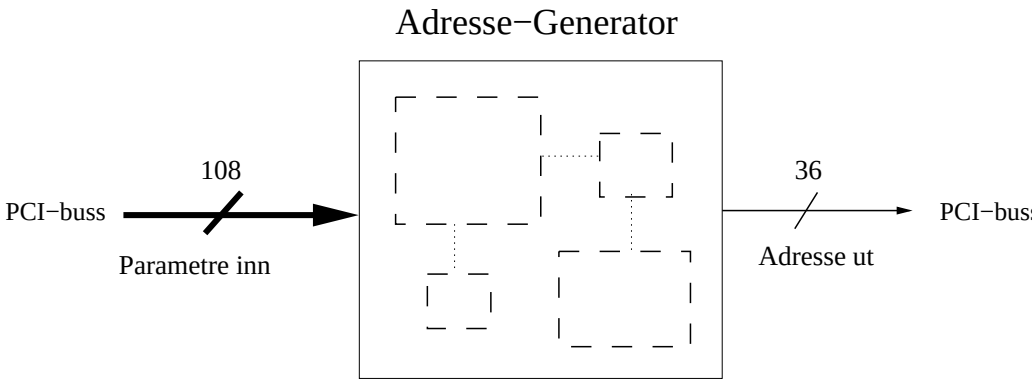
- ***Første (base) adresse :*** Denne parametren angir den første adressen som adress-genatoren skal sende til RAM
- ***N1 :*** Hvor mange adresser som det er i x-retning i avbildningsrommet.
- ***N2 :*** Hvor mange linjer det er i y-retning i avbildningsrommet. Hvis det kun er 1 dimensjon, så vil det være kun 1 linje.
- ***N3 :*** Spesifiserer hvor mange plan det er i z-retning i avbildningsrommet, dersom parametersettet spenner over 3 dimensjoner.
- Δ_1 ***:*** Dersom parametersettet spenner over 2 eller 3 dimensjoner angir denne parameteren hvor stort mellomrommet er mellom hver linje i et plan.
- Δ_2 ***:*** Dersom parametersettet spenner over 3 dimensjoner, så spesifiserer Δ_2 hvor stort mellomrommet er mellom planene i avbildningsrommet.

4.3 Beskrivelse av programmet

Adresse-generatoren er programmert i VHDL og består av et hierarki, hvor det er en toppnivådel som knytter sammen de andre modulene. Denne toppmodulen har en del inn og utganger, hvor de viktigste er innparametrene og utadressen til RAM. Inne i toppnivå modulen er ruting



Figur 4.6: Avbildning og parametersett



Figur 4.7: Enkel oversikt over toppnivået

mellom de forskjellige blokkene gjort. Utgangene til en modul som skal være innganger til en annen modul blir forbundet med ledninger som er deklarerte og tilordnet i toppnivåmodulen, figur 4.7.

4.3.1 Utviklingsmetode og verktøy

Vhsic Hardware Description Language (VHDL) er kun et språk, og for å kompilere koden må man ha en kompilator. Det å kompilerere innebærer å finne syntaktiske feil i koden, slik som gcc gjør med C++ kode. Det er første steg i utviklingen fra kode til en programmeringsfil. På dette nivået kan man simulere koden på et funksjonelt nivå, så sant det ikke er noen syntaktiske feil etter kompilering. I simuleringen kan man finne feil.

Neste hovedsteg er å gjøre om designet til logiske porter, registere etc. Dette kalles for syntese. Etter syntesen vil utviklingsverktøyet kjøre Place and Route. Da blir logikken plassert og rutet på den kretsen man har valgt som mål. Til slutt genererer ISE en programmeringsfil av designet, som blir programmert ned på FPGA'en. Denne blir lagret i ROM'en på kretsen etter at iMPACT har legd et egnet format av den.

Vi bruker Xilinx Integrated Software Environment (ISE) som verktøy. Dette verktøyet kan gjøre hele prosessen som nevnt over, dersom det er en Xilinx krets som brukes. I tillegg knyttes simulatoren Modelsim til ISE. Denne bruker vi til funksjonell simulering. Såkalt høynivå simulering. I løpet av prosessen fra kode til programmeringsfila er klar blir det generert på hvert stadium forskjellige rapporter fra hver del. Disse rapportene sier noe om hastighet, plassbruk etc.

Siste steg er å lage en programmeringsfil som vi laster ned på FPGA'en som er satt på et labkortet. Dette kortet er et testkort for å teste et designet, og blir forklart nærmere i kap. 5.

4.3.2 Toppnivåbeskrivelse

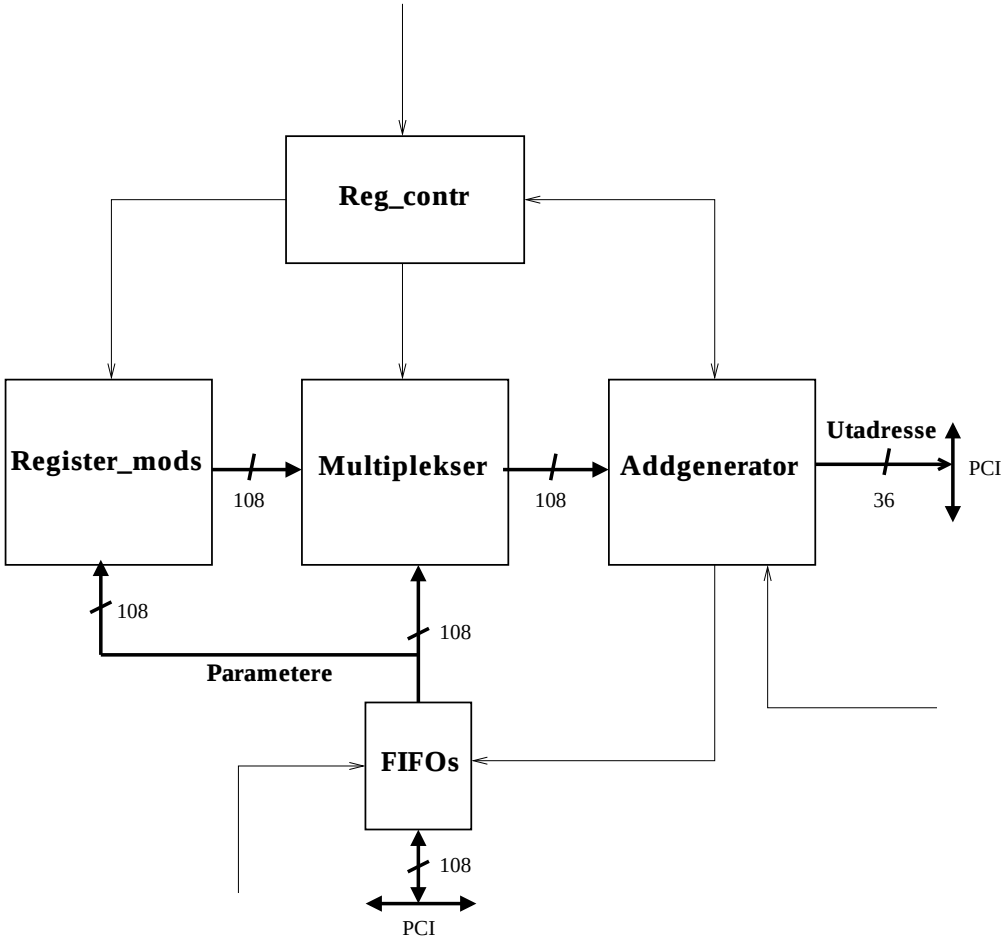
Toppnivåmodulen knytter sammen modulene addgenerator, fifo, reg_contr, multiplex og register_mod. I designet brukes 3 stykker av fifo modulen og også 3 register_mod. Vi ser på figur 4.8 en blokkbeskrivelse av toppnivået. Som vi ser på blokknivået, så kommer innparametrene til adresseringskretsen fra PCI-bussen. Parametrene skrives etter hverandre inn i fifo modulen og legges der i en kø. Dette gjøres ved at parametersettene er tilgjengelige i tid, en etter en på PCI-bussen når styresignalet går høyt for hvert parametersett.

De tynne ledningene på figur 4.8 representerer styresignaler som det er en del av imellom modulene og fra kontroll-logikk utenfra inn til toppnivået. En ledning på figuren kan representere ett eller flere signaler. De er tegnet kun for å illustrere at det er styresignaler imellom moduler etc. Styresignalene kan komme fra applikasjonen som kjører på PC'en eller ifra kontroll logikk som ligger rundt toppnivået. Den kontroll logikken som trengs, kan programmeres på samme FPGA som toppnivået.

Parametersettet leses ved start rett inn i addgenerator modulen. Senere så skrives de til register_mod og ligger klar der til neste bytte av et parametersett. Multiplekseren brukes til å velge om parametrene skal komme rett fra fifo-modulen eller register_mod.

Figur 4.9 viser en skjematisk framstiling av toppnivådelen som ISE framstiller etter syntetisering av vhdl-beskrivelsen av programmet. ISE generer en skjematisk tegning over inn og utganger og rutingen modulene imellom. I tillegg lager den logiske porter, registre, addere og annen kombinatorisk logikk som er deklarert i vhdl-koden. På blokkskjemaet ser vi at alle inngangene ligger til venstre og utadressen, *address_out* ligger helt til høyre. Det er et par utganger fra fifomodulene som ikke brukes, men de er ikke viktige for at adresse-generatoren skal virke på dette stadiet.

I tillegg så ser vi på figur 4.9 alle inn/utganger i hver av modulene. Inngangene i hver modul ligger til høyre og utgangene til venstre. Bredden, dvs. størrelsen på inn/utgangene står etter navnet. Dersom det ikke

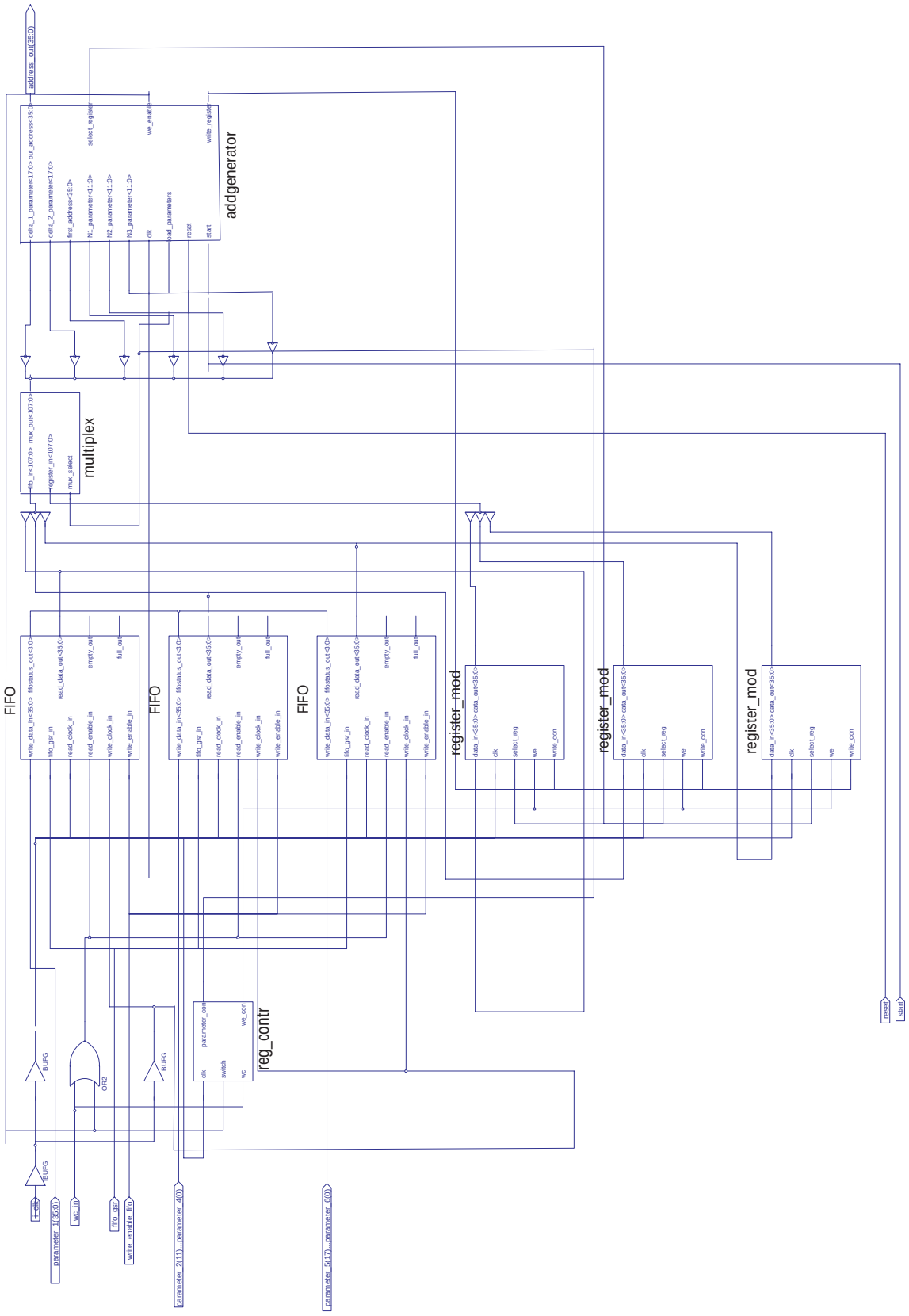


Figur 4.8: Blokkbeskrivelse av toppnivået

står noen størrelse, så er signalet på 1 bit. Videre i dette kapitelet, i beskrivelsen av hver modul, vil vi beskrive inn/utgangene til hver modul. For å se hvorden de er rutet, er det bare å gå tilbake til figur 4.9.

Vi har følgende inngangssignaler til adresse-generatoren på toppnivå som bør forklares nærmere her:

- ***i_clk :*** Det er kun en klokke i designet, og all logikk og alle registre som er klokket, er synkron med denne klokka.
- ***Parameter [1:6] :*** Denne inngangen er på 108 bit og inneholder de 6 parametrene som trengs for å generere adresser fra en avbildning, parametersettet. Parametrene har ikke lik bredde.
- ***fifo_gsr :*** Reset som nullstiller fifo-køene.



Figur 4.9: Skjematisk framstilling av toppnivået

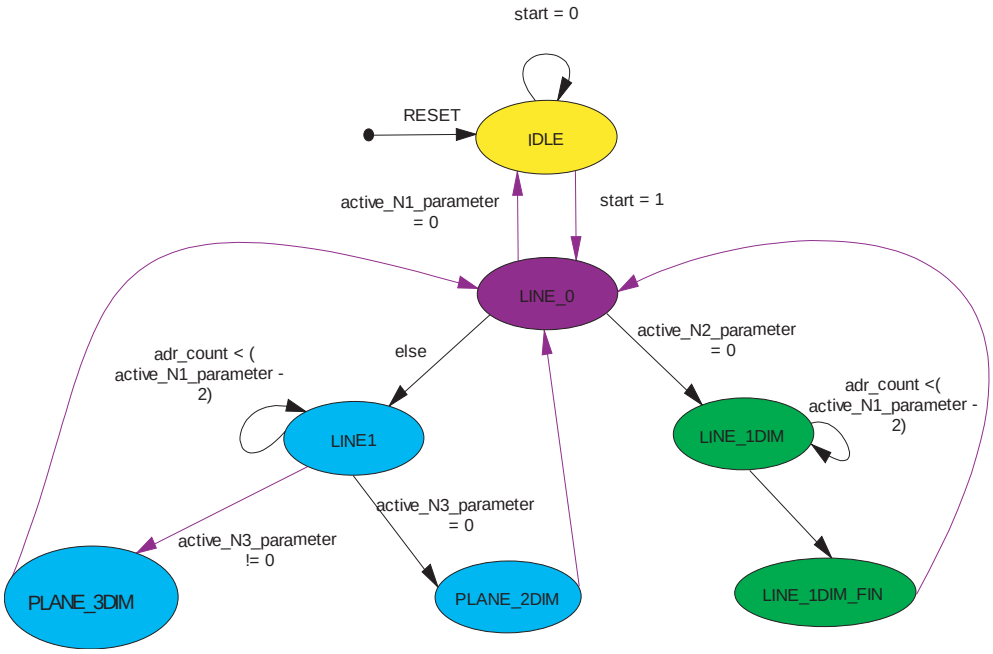
- ***reset :*** Reset som nullstiller adrgen-modulen.
- ***start :*** Dette signalet starter generering av adresser i adrgen-modulen etter at den er nullstilt.
- ***write_enable_fifo:*** Fifo-køene bruker dette styresignalet for å skrive parametre til fifo-køene fra PCI-busen.
- ***wc_in :*** Brukes før oppstart til å lese parametre ut fra fifo-køen til addgenerator modulen, og for å styre multiplekseren i multiplex modulen.

4.3.3 Modulen addgenerator

Dette er den viktigste modulen i hele designet. Det er her parametrene brukes og genereringen av adresser blir gjort. Utgangen fra denne modulen er den genererte adressen, som rutes ut på PCI-bussen til minnet. I tillegg er det utganger som styrer når nye parametre skal hentes inn i fra fifo-køene. Hovedstrukturen i denne modulen består av en tilstandsmaskin, figur 4.10 som sender styresignaler til prosesser som oppdaterer registrene etc. I en av disse prosessene blir utadressen, *address_out*, satt av interne styresignaler fra tilstandsmaskinen. I tillegg så blir tre styresignaler, *we_enable*, *select_register* og *write_register* som går til andre moduler, satt her.

Det er på sin plass å se nærmere på tilstandsmaskinen i denne modulen fordi det er den som er "kjernen" i hele systemet. Tilstandsmaskinen består av 7 tilstander. Default tilstanden er **IDLE**, og i denne tilstanden står maskinen og venter på at *start* går høy. Da starter adressegenereringen i neste tilstand, **LINE_O**. Parametrene er allerede lastet inn ved hjelp av innsignalet *load_parameters* som er et styresignal utenifra til toppnivåmodulen, slik som signalet *start*.

Adresse-generatoren skal kunne jobbe etter en avbildning i 1, 2 eller 3 dimensjoner. I stedet for å ha en egen parameter som angir hvor mange dimensjoner parametersettet spenner over, så blir antall dimensjoner for hvert parametersett satt i N2 og N3 parametrene. Dersom avbildningen er i en dimensjon blir N2 parameteren satt til 0, og i en avbildning i 2 dimensjoner blir N3 parameteren satt til 0. For en 3 dimensjonal avbildning behøves det ikke å gjøre noe spesielt med N1, N2 eller N3, siden 3 dimensjoner er "default". Å sette noen av de overnevnte parametrene, N1 eller N2, til 0, er noe som blir gjort utenfor toppnivået.



Figur 4.10: Tilstandsmaskinen i addgenerator modulen

Alle 6 parametrene ligger i interne registre i modulen og brukes når programmet kjører. I tilstanden **LINE_O** er det en test for å bestemme hvem av de to neste tilstandene som er neste. Testen sjekker om *active_N2_parameter*, som er en av de 6 registrene, er lik 0. Dette registeret bestemmer hvor mange steg i y-retning det er i avbildningen. Dersom den er lik 0, så er avbildningen kun i 1 dimensjon og neste tilstand blir **LINE_1DIM**. Hvis *active_N2_parameter* er forskjellig fra 0, så er neste tilstand **LINE1**. Da er det 2 eller 3 dimensjoner, som parametersettet spenner over.

I tilstandene **LINE1** og **LINE_1DIM** blir adressen plusset på med 1 hver gang. En teller blir også addert med 1. Denne telleren ligger i en prosess utenfor tilstandsmaskinen og lagrer hvor mange steg som er gått i x-retningen. I registeret *active_N1_parameter* er N1- parameteren lagret. Dersom telleren er mindre enn *active_N1_parameter* - 2, så vil neste tilstand være den samme, ellers så vil neste tilstand bli **LINE_1DIM_FINISHED**, hvis 1 dimensjonal avbildning. Neste tilstand etter **LINE_1DIM_FINISHED** blir ellers **PLANE_2DIM** eller **PLANE_3DIM** dersom avbildning i 2 eller 3 dimensjoner.

For å tilfredstille kravet om ingen hull i adressestrømmen, så teller bare telleren i tilstandene **LINE1** og **LINE_1DIM** kun til *active_N1_parameter*

- 2. Den første adressen i hver linje er allerede generert. Ved start er den allerede klar på utgangen ved oppstart, så derfor trengs det kun å telles til N1 - 2, når startindeksen er 0. Ved linjeskift i et plan, fra et plan til et neste, eller ved et nytt parametersett, så blir utadressen plusset på eller satt til en ny verdi. Dette gjøres i tilstandene **LINE_1DIM_FINISHED**, **PLANE_2DIM** og **PLANE_3DIM**.

Ved å ha neste parametersett klart ved et bytte, vil det gjøre at et bytte ikke gir noen hull i adressestrømmen. Når genereringen av et sett er ferdig, blir registeret til utadressen satt til den nye start adressen for det neste settet; utadresse <= Første adresse.

Hva blir situasjonen ved oppstart når første parametersett skal genereres, da det ikke har vært noen generering før som har tilordnet førsteadresse? Dette løses ved styresignalet *load_parameter* som forklart over.

I tilstanden **LINE_DIM_FINISHED** er det den ene linjen som utgjør en avbildning i 1 dimensjon, som er blitt ferdig generert. Neste parametersett er allerede klart, og lagret temporært, og et styresignal setter de aktive registrene, de som brukes, lik de temporære. Utadressene blir her satt lik parameteren *first_address*. Neste tilstand fra denne blir **LINE_0** som alltid er første tilstand i starten av et nytt parametersett.

Dersom avbildningen er i 2 dimensjoner, så er neste tilstand etter **LINE1**, **PLANE_2DIM**. Her blir adressen addert med $\Delta 1$ dersom det er flere linjer i planet i avbildningen å traversere, ellers vil et nytt parametersett bli brukt ved neste klokkepuls i tilstanden **LINE_0**.

I en avbildning i 3 dimensjoner blir logikken mer kompleks. Tilstanden etter **LINE1** blir da **PLANE_3DIM**. I denne tilstanden kan utadressen legges til med 3 forskjellige verdier. Dersom det er flere steg i y dimensjonen, og vi er i planet, så vil utadressen legges til med $\Delta 1$. $\Delta 2$ vil bli lagt til utadressen når planet er ferdig , og neste utadresse er første adresse i neste plan i kuben (rektanglet). Når siste linje i det siste planet er ferdig, vil et nytt parametersett bli brukt i neste tilstand, **LINEO**. Utadressen blir da i tilstanden **PLANE_3DIM** satt til *first_address*, akkurat som bytte av parametersett i tilstandene **LINE_1DIM_FINISHED** og **PLANE_2DIM**.

4.3.4 Modulen Fifo

Hvert parametersett ligger i Fifo-køer. Fifo-modulen er en ferdiglagd først inn, først ut kø, som importeres i et bibliotek fra Xilinx. Hvert element i køen har en bredde på 36 bit, og hver kø har plass til 511 elementer. For å lagre elementene, brukes dual-port blokk RAM som kan brukes med en klokkehastighet på 200 MHz.

For å skrive til køen settes innsignalet, *write_enable_in* høy på stigende klokkeflanke. På neste klokkepuls, når klokka stiger, må verdien ligge på innbussen *write_data*. Lesing fra køen foregår på samme måte. Innsignalet *read_enable_in* går høy, og ved neste stigende flanke er verdien klar på utgangen, *read_data_out*.

Det er 6 parametre i hvert parametersett. Hver parameter har ikke sin egen fifo-kø, men de ligger i 3 køer. Parameteren *first_address* har sin egen kø. Den andre fifo-køen deles av dimensjonstellerne *N1, N2 og N3*. Hver av disse 3 parametrene får en størrelse på 12 bit. Den tredje og siste køen deles av 2 parametre, $\Delta 1$ *og* $\Delta 2$, som er på 18 bit hver.

4.3.5 Modulen reg_contr

Denne modulen har kun 2 innganger, *switch* og *wc.switch* er et styresignalet fra adr_gen. Dette styresignalet går også til Fifo-køene og leser et element fra dem. Reg_contr tar og forsinker det signalet slik at pulsen er forskjøvet en klokkesykel i tid. Signalet er kun høyt i en klokkeperiode.

Det er to utganger i reg_contr. *parameter_con* er et styresignal for multiplekseren og *we_con* er det forsinkede signalet som er beskrevet over. Modulen reg_contr består av kun en tilstandsmaskin med 3 tilstander.

4.3.6 Modulen register_mod

Parametersettet blir mellomlagret i denne modulen fra fifo-køen og til addgenerator modulen. Det er 2 registere i denne modulen som hver kan inneholde et parametersett, på 108 bit. Mens ett parametersett brukes, blir det neste hentet fra fifokøen og legges her, klart til bruk i adrgen modulen.

Det er 3 innganger til denne modulen, som er utganger fra adrgen modulen. Inngangen *write_con* bestemmer hvilket register som det skal skrives til. Det forsinkete, *we*, signalet fra reg_contr bestemmer når det skal skrives til registeret. Det går høyt samtidig med at parametersettet er klart på fifo-utgangen og skrives til et av registrene. Det siste innsignalet *select_reg* styrer en multiplekser inne i denne modulen, som velger hvilket av registrene sine utganger som skal gå til addgenerator.

4.3.7 Modulen multiplex

Dette er en veldig enkel modul. Det eneste den gjør er å styre hvilket parametersett som skal inn til addgenerator modulen. Ved oppstart er det ingen pararametre som ligger i registene. Det er ikke nødvendig å skrive parametrene først til et register og så videre til adrgen. I stedet blir parametersettet rutet rett inn i adrgen ved oppstart, ellers så styrer denne multiplekseren parametrene fra registrene i register_mod til addgenerator. Styresignalet som bestemmer hvor parametersettet til addgenerator modulen skal rutes ifra, blir valgt i reg_contr modulen.

Kapittel 5

Eksperimenter

Vi vil her presentere simuleringsresultater på toppnivå som viser hvordan designet virker og oppfyllelsen av noen av kravene i fra kap. 4.1.1. I tillegg vil viktige resultater som hastighet og plassbruk bli gjennomgått, samt testing på FPGA'en på labkortet.

5.1 Simuleringsresultater på funksjonelt nivå

En stor del av tiden i utviklingen av et design går med på å simulere og teste at designet stemmer på funksjonelt nivå. Dette vil si at man fjerner semantiske feil i koden og forandrer logikk slik at designet stemmer overens med krav og føringer som er gitt. Ved hjelp av simuleringsplott er det lettere å illustrere hvordan kretsen virker. Vi vil vise 3 utdrag fra en simulering av toppnivået: startfasen m/skriving til fifo-køen, midt i gjennomkjøringen av et parametersett og ved et bytte av parametersett.

Den første delen av simuleringen er fra starten av kjøringen, figur 5.1. Inngangene til toppnivået er de 12 øverste kurvene på plottet, fra *i_clk* til og med *fifo_gsr*. Etter *fifo_gsr* er den eneste utgangen: *address_out*.

Resten av signalene er interne registere i modulen addgenerator. Registeret *address_out* er en teller som teller hvor mange adresser som er generert i X-retningen i avbildningsrommet. *Y_count* og *z_count* er tellere for hvor mange steg i y og z-retning i avbildningsrommet som er traversert. Registrene *active_N1_parameter*, *active_N2_parameter*, *acti-*

ve_N3_parameter inneholder de 3 parametrene N1, N2 og N3 som er nevnt i kap. 4.2.5. Det neste parametersettet ligger i registere og hver parameter har sitt register. N1, N2 og N3 for neste gjennomkjøring er lagret i de 3 neste signalene *n_1_parameter_temp*, *n_2_parameter_temp* og *n_2_parameter_temp*.

De forskjellige tilstandene i tilstandsmaskinen som er beskrevet i kap. 4 ligger i tilstandsregistre. Hver tilstand har hvert sitt register, fordi syntheseverktøyet optimaliserer for hastighet og bruker derfor one-hot-encoding. Nåværende tilstand er signalet *present_state*. Det vil si hvilken tilstanden som tilstandsmaskinen er i ved tiden t. Ved neste klokkepuls, t + Δt, vil nåværende tilstand bli lik det neste-tilstands-registeret *next_state* er ved tiden t.

5.1.1 Start

Første utdrag av simuleringen viser startfasen. På plottet 5.1 er det 3 tidspunkter som bør forklares nærmere:

- ***Skriving til fifo-køen (1):*** Ved (1) blir det første parametersettet lest inn i fifokøene. De seks parametrene er innsignalene *parameter_1* til *parameter_6*. Parameteren første adresse er signalet *parameter_1*. Videre vil hver parameter følge i den rekkefølgen de er beskrevet i kap. 4.2.5: N1, N2, $\Delta 1$ og til slutt $\Delta 2$ er signalet *parameter_6*.

  Det første parametersettet spenner over en dimensjon på 3. Det neste settet blir skrevet til køene ved 85 ns og er på 1-dimensjon med kun 15 adresser (*parameter_2*). De andre parametrene i dette settet er satt til 0. Spesielt må N2 være lik 0. Jfr. forklaringen i kap. 4.3.3 for en avbildning i 1 og 2-dimensjoner. Neste parametersett som skrives, ved 100 ns, spenner over 2 dimensjoner og i dette settet er N3-parameteren satt til verdien 0. Det siste settet er i 3-dim og tatt med for å vise at transisjonene mellom parametersett med forskjellige dimensjoner ikke gir hull i adressestrømmen; adresse-generatoren virker korrekt.

- ***Skriving til parameterregistrene (2):*** Signalet *wc_in*, som er et styresignal utenifra, går høyt ved 140 ns. Dette signalet styrer skrivingen til parameterregistrene før oppstart slik at disse registrene er oppdatert og klare. Ved 145 ns går klokka høy og prossesen som

styrer oppdateringen av parameterregistrene er synkron og vil derfor kjøres. Inne i denne prosessen vil en '1' på *wc_in* "registreres", og registrene vil først bli oppdatert først ved neste klokkecykel (2).

Av plasshensyn, så er ikke parameterregistrene for parametrene $\Delta 1$ og $\Delta 2$ tatt med. De 2 parametrene ligger i registrene *active_delta_1_parameter* og *active_delta_2_parameter*. Disse blir oppdatert likt som de andre parameterregistrene. Under oppstartfasen brukes ikke de temporære registrene som lagrer neste parametersett i addgenerator modulen. Siden disse ikke brukes har de verdien verdien 'X' helt til neste parameterbyttefase innledes.

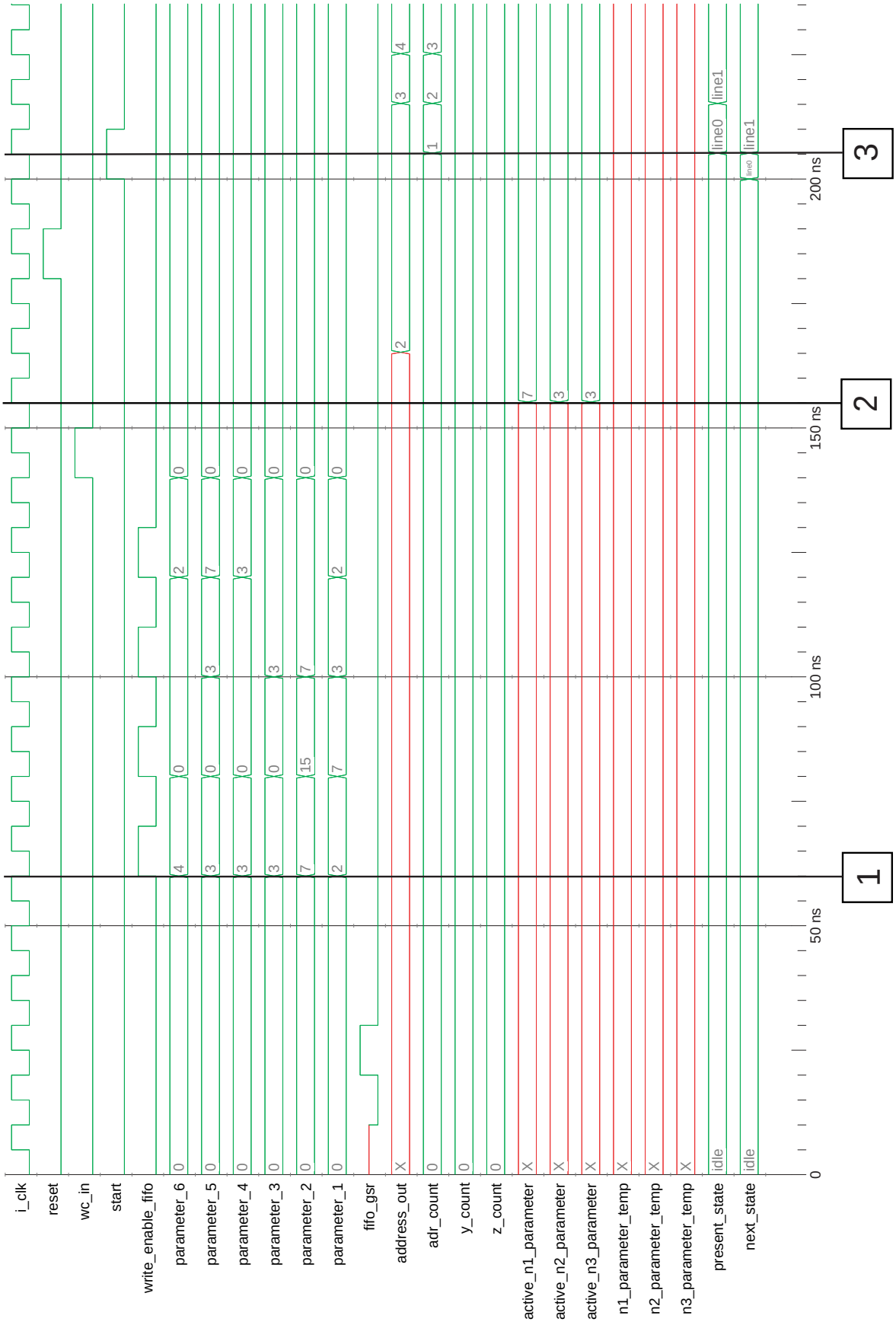
- ***Oppstart (3):*** Signalet *start* går høy ved 200 ns. Tilstanden da er **IDLE**. Oppdateringen av registereret *next_state* er asynkron og vil med en gang bli satt til line0 ved 200 ns. Siden oppdateringen av nåværende tilstand (*present_state*) er synkron med klokka vil **LINE0** være nåværende tilstand ved (3). Da har adressegenereringen startet og første adresse er allerede klar på utgangen *address_out*. Ved neste klokkecykel på 210 ns er nåværende tilstand blitt **LINE1** og en ny utadresse er generert: 3. Slik vil den stå i **LINE1** helt til den har generert 6 adresser, like mange som parameteren N1 - 1.

5.1.2 Generering av adresser i 3 dimensjoner

Første parametersett er i 3 dimensjoner og starter generering ved 210 ns på figur 5.1. Fortsettelsen er vist i plottet 5.2 og viser ett par viktige overganger i avbildningsrommet i et 3D-parametersett.

- ***Overgangen fra en linje til neste linje (1):*** Ved (1) har det blitt generert 2 linjer og den andre linjen er ferdig. Telleren for antall adresesser i linjen, *adr_count* starter på 0 og har telt til 5 på forrige klokkecykel. Det er da generert 6 adresser i den linjen i tilstanden line1. I tillegg til den første adressen i hver linje blir dette 6 + 1 = 7. Dette stemmer overens med N1 parameteren, *parameter_2*, for dette settet som er 7.

  Tilstanden i (1) er **PLANE_3DIM**. Registeret *y_count* teller hvor mange linjer i planet som er traversert og er 1, (indekseringen starter på 0) som totalt er 2 linjer i planet. I **PLANE_3DIM** blir dette registeret plusset på med 1. Det blir først oppdatert en klokkecykel etter ved 345 ns.



Figur 5.1: Plot av startfasen m/ skriving til fifo-køen

I tilstanden **PLANE_3DIM** blir adressen plusset på med $\Delta 1$. Utadressen ligger i et eget register som blir oppdatert etter en klokkecykel. Utgangen *address_out* er satt lik dette registeret og det tar en klokkesycel til før den første adressen i neste linje er klar på utgangen. Dette skjer ved (2), hvor *address_out* er blitt 20. Dette stemmer med $\Delta 1$ parameteren for dette settet som er 3. I tilstanden **PLANE_3DIM** blir derfor adresseregisteret tilordnet verdien 17 + 3 = 20.

- ***Overgangen fra et plan til neste (3):*** Fra ett plan til det neste er også tilstanden plane_3dim i bruk ved overgangen, slik som imellom linjene i planet. Registeret *y_count* er 2, og siden indekseringen starter på null, så har det blitt generert 3 linjer. Parameteren N3 er i dette parametersettet 3, og den neste nye adressen som skal genereres i tilstanden **PLANE_3DIM** er i neste plan.

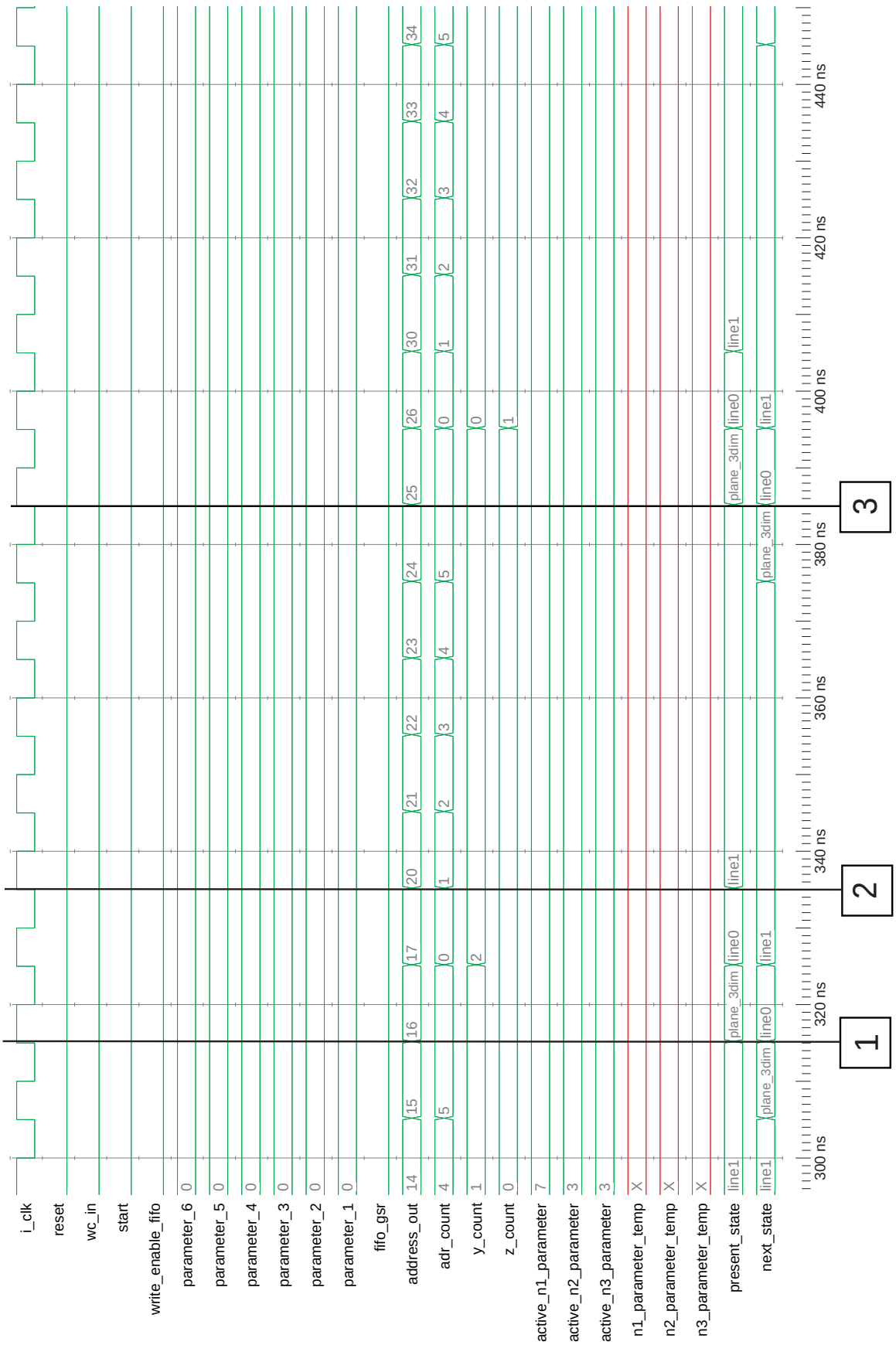
  Ved neste klokkesykel ser vi at registeret *y_count* er satt til 0, siden vi starter på et nytt plan. Registeret som teller antallet plan er *z_count*. Dette registeret blir plusset på med 1 ved 395 ns.

  2 klokkecykler etter (3), ved 205 ns, blir *address_out* 30. Dette er første adressen i det andre av totalt 3 plan. Adressen ved forrige klokkecykel er 26, som er siste adresse i det første planet. Adresseregisteret har blitt addert med 4 i tilstanden **PLANE_3DIM** i (3), som gir adressen 30. Dette stemmer overens med $\Delta 2$ som er 4 i dette parametersettet.

5.1.3 Bytte av parametersett

Et bytte mellom parametersett innebærer at registrene som inneholder parametrene oppdateres på riktig tidspunkt, og at det neste parametersettet må være klart for en slik operasjon. På plottet 5.3 ses den viktige fasen hvor dette skjer.

- ***Oppdateringen av parameterregistrene (1):*** I god tid før det aktuelle byttet av funksjonen, dvs. parametersettet, blir et styresignal sendt ifra modulen addgenerator til fifokøen. Det neste parametersettet blir lastet ned i modulen register_mod. Ved (1) blir neste parametersett lagret midlertidig i modulen addgenerator i tempregistrene. Registeret *n1_parameter_temp* inneholder n1 parameteren for neste sett og er 15 ved (1). Parameteren n2, *n2_parameter_temp*, for neste sett er 0, fordi dette settet kun spenner over 1 dimensjon.



Figur 5.2: Utdrag fra gjennomkjøringen av et parametersett i 3dim

- ***Bytte av parametersett (2):*** Siste gang parametersettet med 3 dimensjoner er i plane_3dim, blir alle de aktive parameteregistrene byttet ut og satt lik de temporære parameterregistrene hvor neste sett ligger klart. Det aktuelle byttet av parametere skjer her, men når det gjelder tilstander, så er den aller første tilstanden i det nye 1-dimensjonale settet ved neste klokkesycel, på 815 ns. Tilstanden vil alltid bli line0 etter bytte av parametersett.

  Det tar en klokkecykel etter (2) før det nye parametersettet er oppdatert i parameterregistrene. Vi ser på plottet at *active_n1_parameter* er da blitt 15. Det holder at de nye parametrene er klare da, siden de først blir brukt i tilstanden line0 for å bestemme neste tilstand.

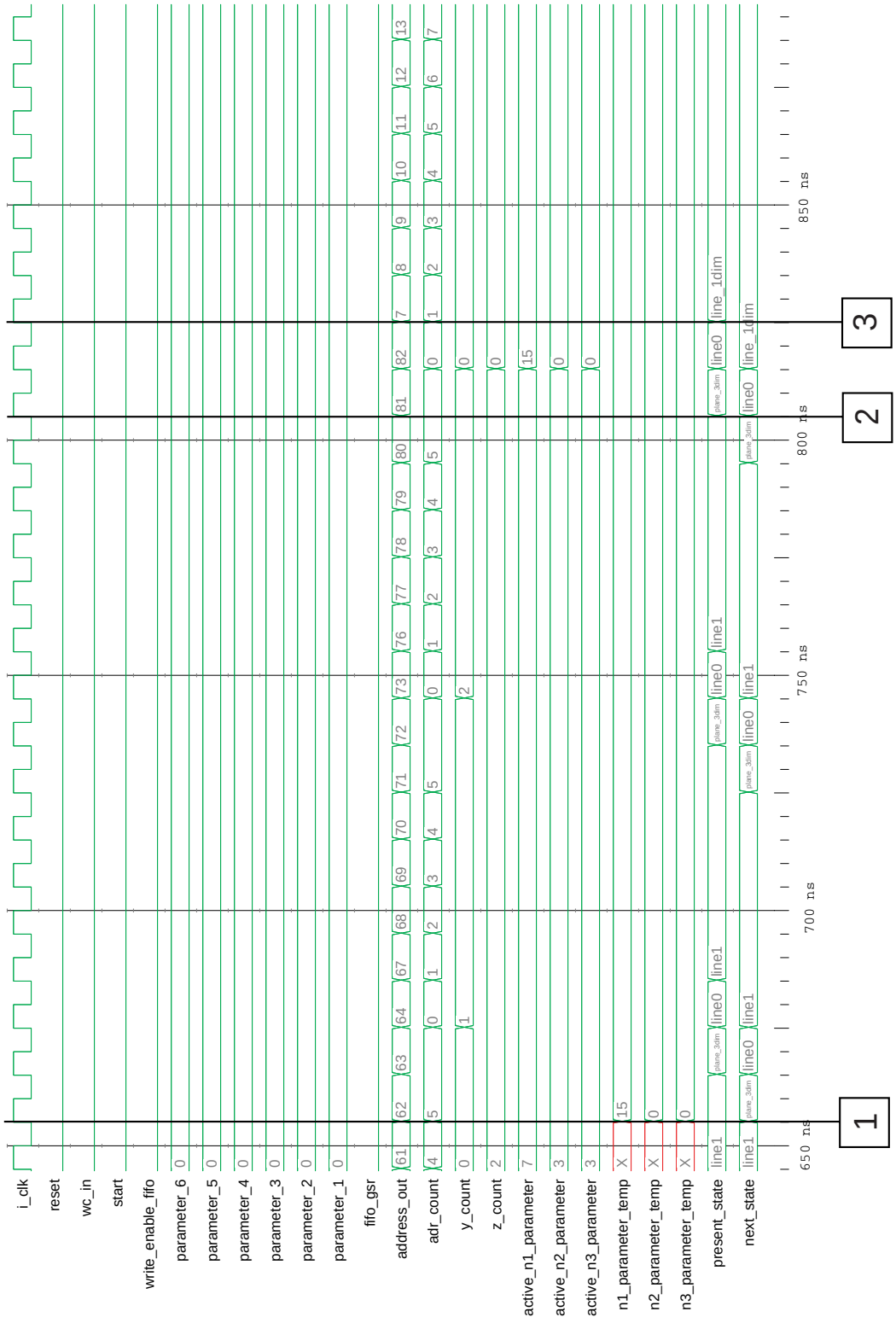
  Siden parametersettet er i 1-dimensjon, blir tilstanden i (3) **line_1dim**. Her vil tilstandsmaskinen stå i denne tilstanden og generere 15 - 1 = 14 adresser. Den første adressen i 1dim settet vil i (3) være klar på utgangen og sendt til minnet. Som vi ser så er første adresse 7, og det stemmer med verdien til *parameter_1* i det andre settet som skrives til fifokøene på figur 5.1 ved 80 ns.

5.2 Ressursbruken til designet på FPGA'en

Neste steg etter simuleringen er å kjøre videre igjennom prosessen som nevnt i kap. 4.3.1. Vi optimaliserer for hastighet i ISE, slik at plassbruken vil bli noe større. Det er nok av plass på en Virtex-II pro til å gjøre det.

5.2.1 Ressursbruk

Vi har samlet en del estimeringer og resultater fra syntese-rapporten og place and route rapporten i tabell 5.2.1. Det kan også være aktuelt å bruke andre kreser fra Xilinx som har andre egenskaper enn Virtex-II pro. Disse kan ha flere I/O-blokker, være tregere, ha færre porter etc. Pga. at vi bruker en ferdiglagd fifo-kø som bruker blokkram som er spesiell for Virtex-II pro og Virtex-II, er det vanskelig å overføre denne køen i designet til andre Virtex familier som Spartan og Virtex-E. Tabellen inneholder derfor designet implementert på Virtex-II pro og en Virtex-II til og med place and route. Under implementasjonen på de andre kretsene er derfor kun syntese stadiet som har vært mulig å gjennomføre.



Figur 5.3: Bytte av parametersett

Hastighet-1 i tabellen er kun et estimat på maks. klokkefrekvens som er hentet fra rapporten etter syntesen. De to andre verdiene, IOB (Input Output block) og antall slices som designet bruker, framkommer også i rapporten etter syntesen. Den siste verdien i tabellen, hastighet-2 er den estimerte klokkefrekvensen etter place and route. Dette estimatet er mer nøyaktig enn estimatet i synteserapporten.

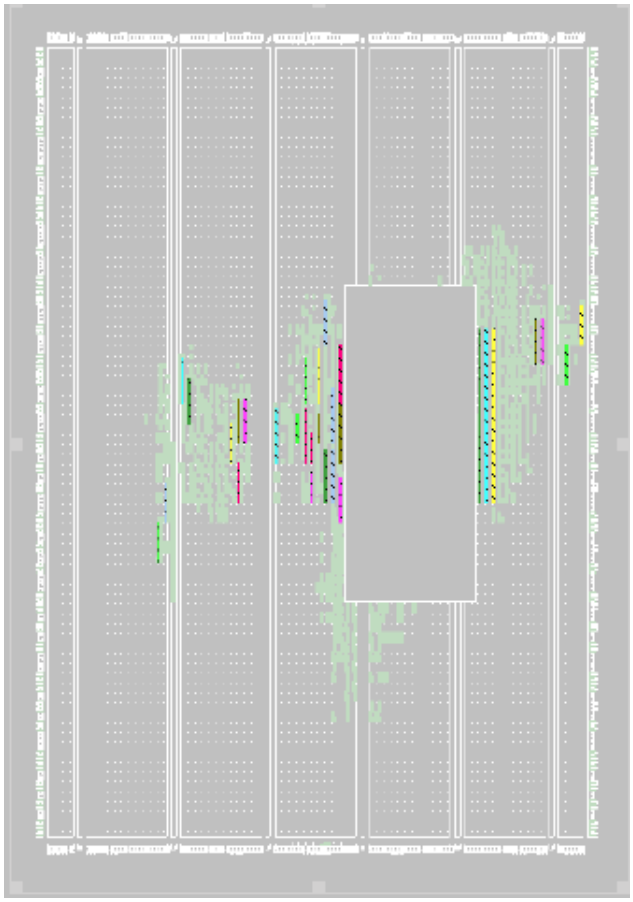
krets	pakke	speed grade	IOB (%)	slices (%)	hastighet1 (MHz)	hastighet2 (MHz)
xv2vp7	fg456	-7	60	12	141	144
xv2vp7	fg456	-5	60	12	108	108
xv2vp7	fff672	-7	37	12	141	143
xv2v1000	fg456	-6	46	11	129	136
xcv300e	fg456	-8	47	19	72	—
xcv1000	fg680	-6	29	5	59	—
xc3s5000	fg900	-4	23	1	83	—

Tabell 5.1: Resultater fra implementasjonsprosessen

Det finnes et stort antall pakker og kretser i familiene Virtex-II og Virtex-II pro. Forskjellen mellom kretsene i familiene kan være antall I/O blokker, slices, blokkram etc. I Virtex II-pro har f.eks den "råeste" kretsen i denne familien 1164 I/O-blokker og over 99000 slices. En oversikt over spesifikasjonene til noen Virtex familier finnes på [28].

Det er et stort antall kretser i hver familie, så har vi implementert kun et fåtall. Det er kun hensiktsmessig å ta med resultater i forbindelse med implementering av kretsen som vi har tilgjengelig på laben, xv2vp7, når det gjelder Virtex-II pro familien. Alle implementasjonene har satt hastighet som optimaliseringsmål. Dette vil si at høyest mulig klokkefrekvens er målet, og de prosserer i prosesskjeden som kan prioritere dette gjør det, framfor å prioritere plassbruk.

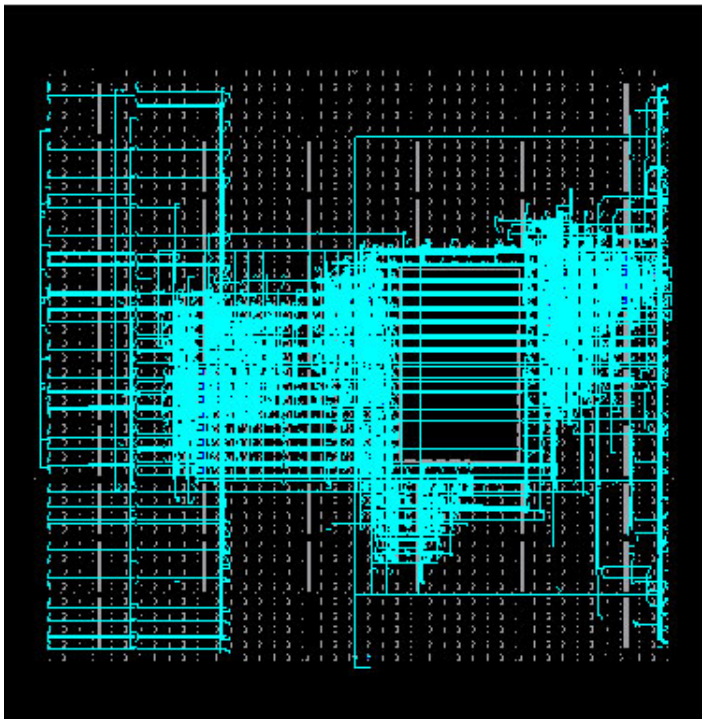
Vi ser på tabellen at den implementasjonen med høyeste mulig klokkefrekvens er utført på kretsen som er tilgjengelig på laben med en speed grade på -7. Speed grade er lik propagasjonsforsinkelsen (propagation delay) gjennom en CLB. Et større negativt tall betyr at kretsene er raskere. Den største frekvensen er etter place and route estimert til ca 144 MHz. På eldre kretser, som Virtex-E, Virtex og Spartan ser vi at de på langt nær kan vise til de samme hastigheter som Virtex-II og Virtex-II pro kan. Transistorstørrelsen er ofte mindre på de nyeste familiene. De eldste familienen vil rett og slett være for trege for dette designet.



Figur 5.4: Plasseringen av logikken på xv2vp7

En prioritering av plassbruk innebærer at designet skal bruke færrest mulig slices. Det har vi ikke prioritert i prosessen, men det er uansett nok av plass på labkretsen. Det blir bare brukt 12 % av slicene på FPGA'en som er tilgjengelig på laben, og det er ikke en stor utnyttelse av plassen. Vi vil i neste kapitel presentere en løsning for bedre å utnytte plassen på lab-kretsen.

Den sparsomme plassbruken til designet ser vi på figur 5.4 hvor plasseringen av logikken er gjort på lab-kretsen. En stor del av plassen på kretsen er ruting imellom de logiske blokkene på FPGA'en og IO-blokkene. Som vi ser på figur 5.5 så er det mange forbindelseslinjer som trengs og heldigvis er det verktøyet som gjør rutingen for oss„ selv om man kan gjøre det selv i Xilinx Floorplanner. Det store tomme området på figurene 5.4 og 5.5 er hvor IBM Power CPU'en er plassert.



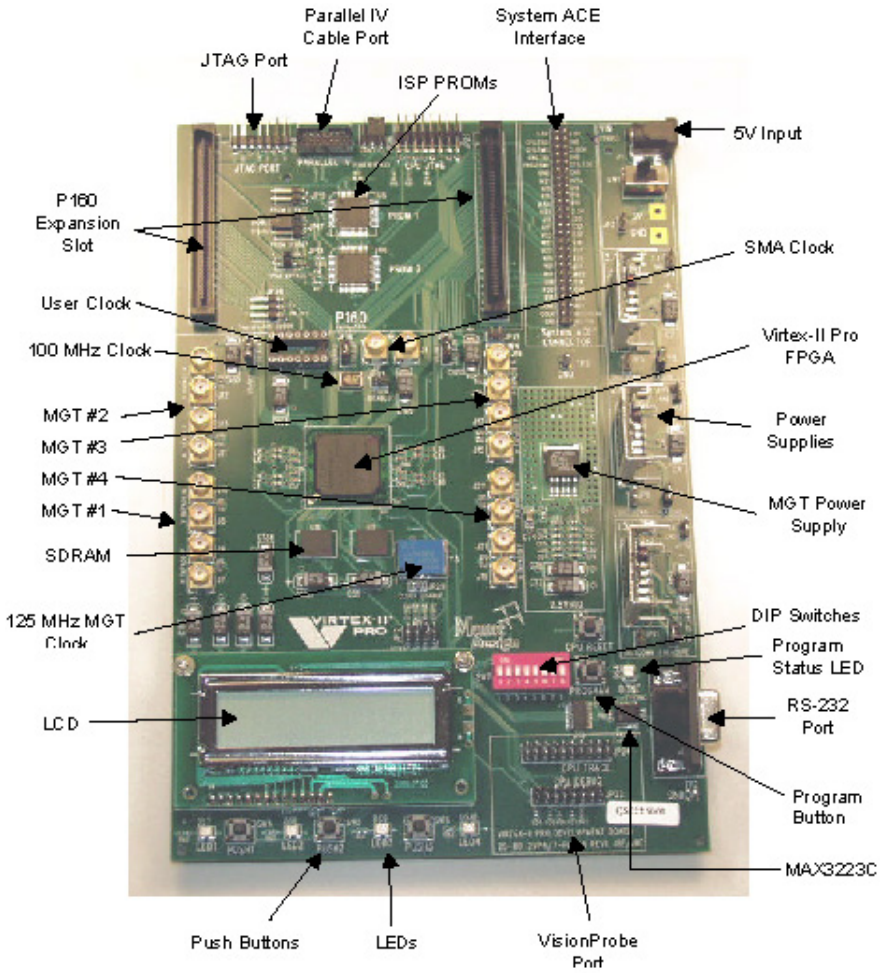
Figur 5.5: Rutingen til designet på xv2vp7

5.3 Funksjonell test av adresse-generatoren på labkortet

I tillegg til den funksjonelle simuleringen av hele designet har vi også testet den viktigste modulen i designet på labkortet, addgenerator, som er beskrevet i foregående kapitel, 4.3.3, men med noen modifiseringer. Hensikten med å programmere og teste denne modulen på kortet er å verifisere at den virker i praksis, ikke bare i simulering, og at genereringen av adresser er riktig. Det er altså kun en funksjonell test som utføres, hvor ting som hastighet etc, ikke blir testet.

5.3.1 Labkortet

Labkortet eller utviklingskort (Development board) som det også kalles er lagd av Memec Design [34], figur 5.6. Det eneste på kortet som er lagd av Xilinx er selve FPGA'en som er plassert midt i. Som vi ser så er det mange muligheter på kortet, men vi bruker kun noen få av dem.



Figur 5.6: Labkortet fra Memec Design [29]

Programmeringen fra PC'en til kortet bruker *Parallell IV Cable port* som med en parallell kabel komuniserer med PC'en. Gjennom denne kabelen blir programmeringsfilene til ROM'ene, *ISP ROM's*, sendt. Vi bruker også *LED*, *CPU Reset* og *Push Buttons* i testen.

5.3.2 Virkemåten til testdesignet

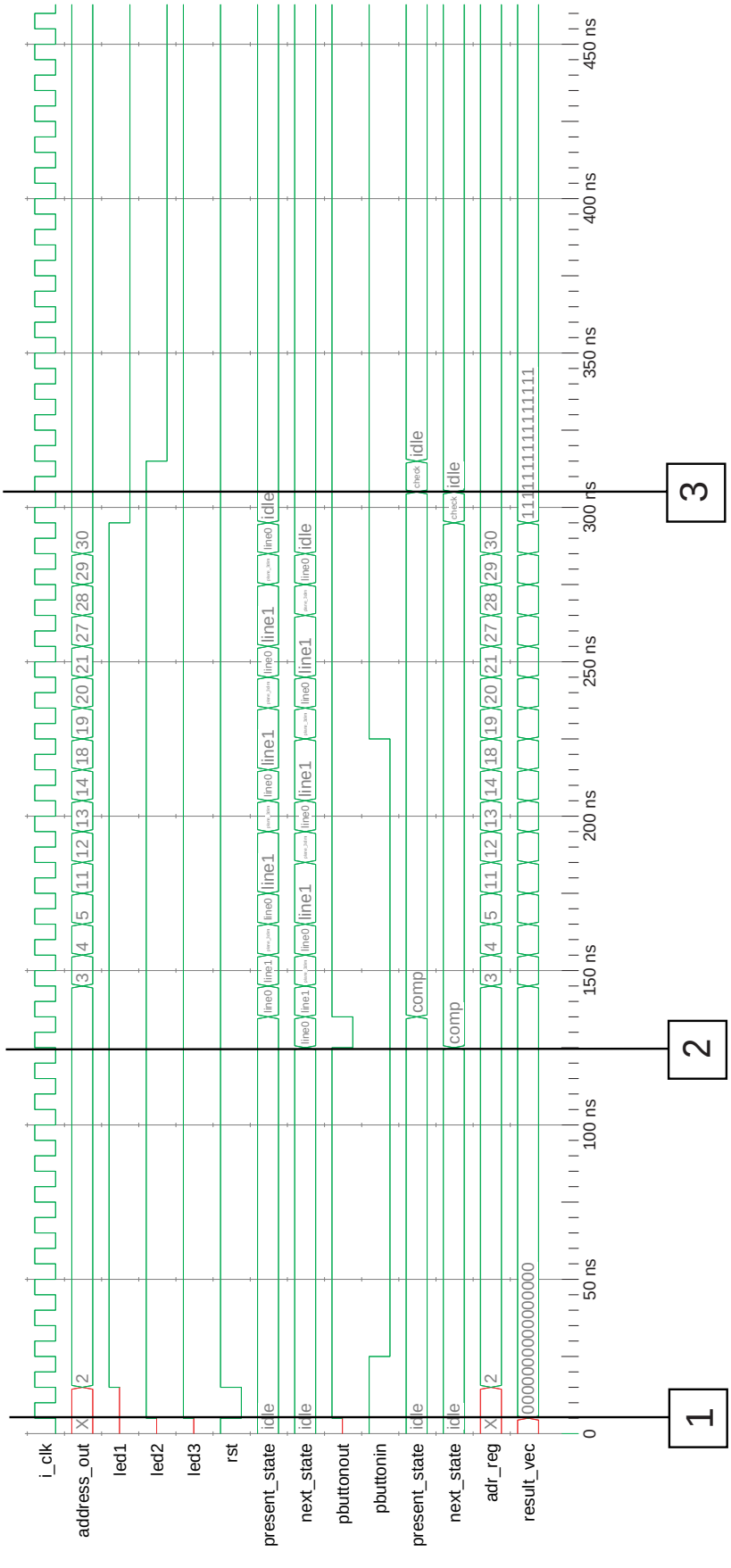
Testdesignet skal kjøre igjennom et parametersett og teste om adressene som blir generert av modulen addgenerator er riktige. For å teste om adressene er korrekte, så er de riktige adressene hardkodet i registere i testdesignet. Vi vet hva de riktige adressene er, fordi de riktige

adressene blir generert i den funksjonelle simuleringen av addgenerator modulen først.

Første steg er å resette alle registrene. Dette gjøres ved å trykke inn *CPU Reset* på kortet. Denne knappen sender ut en puls som går lav og den er en klokkepuls lang. Neste steg er å starte genereringen av adresser. Dette gjøres ved å trykke in *PUSH1* på kortet. Dette vil lage en kort puls på en klokkecykel som starter tilstandsmaskinen i addgeneratormodulen. I tillegg så starter den en sammenligning av de genererte adressene fra modulen med de som er hardkodet i registre. Sammenligningen skjer i sann tid.

Vi ser bedre virkemåten på figur 5.7., hvor følgende punkter i testdesignets gjennomkjøring i funksjonell simulering er viktige:

- ***Reset (1):*** Det finnes ingen simuleringsmodell for *CPU Reset*, så i denne simuleringen påtrykker vi stimuli i testbenken for denne knappen. Signalet ut i fra *CPU Reset* er aktivt høyt på kortet, og ved et trykk på knappen blir et signal sendt ut i en klokkepuls. I simuleringen skjer dette ved 5 ns.
- ***Startingen av adressegenerering (2):*** Trykknappen blir trykt inn ved 20 ns. I trykknapp modulen er det en "debouncer" som teller hvor lenge knappen er trykt inn og venter et visst antall klokkesykler med å sende et signal. Dette signalet er på en klokkecykel, blir sendt ut ved (2) og starter to tilstandsmaskiner. Den ene er adressegenereringen i addgeneratormodulen og den andre er en tilstandsmaskin i toppnivået som sammenligner adresser. En klokkecykel etter (2) starter sammenligningen i tilstanden **comp**. Dersom sammenligningen gir riktig svar, så vil et bit i registeret *res_vec* få en ener. Hvilket bit i *res_vec* som blir satt til 1, er styrt av en teller som teller antall adresser som blir generert. Antallet adresser som skal genereres vet vi på forhånd ut fra simuleringen.
- ***Den siste verifiseringen av adressene (3):*** Etter at tilstandsmaskinen i toppnivået har sammenlignet alle adressene og lagt resultatet i *res_vec* blir tilstanden ved (3) **check**. I denne tilstanden blir alle bitene i *res_vec* sjekket om de er 1. Dersom alle bitene er 1, vil *LED2* gå lav og LED2 på kortet vil lyse grønt. Hvis det er en 0 i ett eller flere av bitene i *res_vec*, så vil istedet *LED3* lyse rødt. En klokkepuls før (3), ved 295 ns, vil *LED1* gå lav og LED1 på kortet vil lyse. Dette betyr at adressegenereringen er ferdig.



Figur 5.7: Funksjonell simulering av testdesignet

5.3.3 Resultatet av testen

Vi fullførte hele prosessen i Xilinx ISE, fra syntese, place and route etc, på testdesignet og tilordnet pinnene på FPGA'en til de riktige ledningene på kortet i UCF filen. Til slutt, etter place and route i ISE, blir det lagd en bitstrømsfil som brukes i Xilinx iMPACT. Dette programmet lager programmeringsfiler av bitstrømmen for ROM'ene på kortet av bitstrømmen.

Designet ble testet etter programmering, og LED1 og LED2 lyste etter og ha utført stegene i henhold til virkemåten til testdesignet . Konklusjonen er at adresseringen virker med et datasett uten parameterbytte med en klokkehastighet på 100 MHz, spesifisert i UCF-filen.

Til slutt lagde vi en feil i testdesignet for å teste om testmodell er til å stole på og om sammenligningene er riktige. Vi satte den første adressen i registeret til å være 0 og ikke 2 som er riktig. Dette gjør at alle bitene i *test_vec* ikke blir 1; LED3 lyste.

Kapittel 6

Diskusjon/analyse

Vi har i de foregående kapitlene presentert konstruksjon og testing av en adresse-genererator som kan generere adresser med gitte parametersett. Et slikt design kan forbedres og utvides i det uendelige. Å gjøre dette for tidsrammen til hovedfagprosjekt begrenser denne muligheten, og løsningen på problemstillingen i oppgaven er en basis for videre arbeid og utvidelse. Vi vil i dette kapittelet se på løsninger for å redusere hastighet, plassbruk og til slutt videre arbeid.

6.1 Hastighet/regnekraft

Som nevnt før er hastigheten til adressegeneratoren viktig for hvor stor regnekraften til beregningsenheten blir. Vi ser på tabellen i kapittel 5.2.1 at vi kan oppnå en klokkehastighet opp mot 144 MHz på den raskeste kretsen. En utvidelse med kontrollgikk og videre utvikling av designet som er presentert i kap. 4, vil redusere hastigheten noe. Vi vil gå ut i fra den høyeste hastigheten som er 144 MHz i våre beregninger av regnekraften.

Utgangen fra designet er 36 bit som rutes inn på adressebussen til minnet. Databussen ut i fra minnet er 64 bit slik at beregningskretsen får ideelt sett en regnkraft på:

$$64 * 144MHz = 64 * 144Mbit/s = 9216Mbit/s = 1152MB/s \quad (6.1.0.1)$$

Virtex-II pro støtter PCI-X standarden [36] som er en bus standard med en overføringsrate på 64 bit ved 133 MHz.

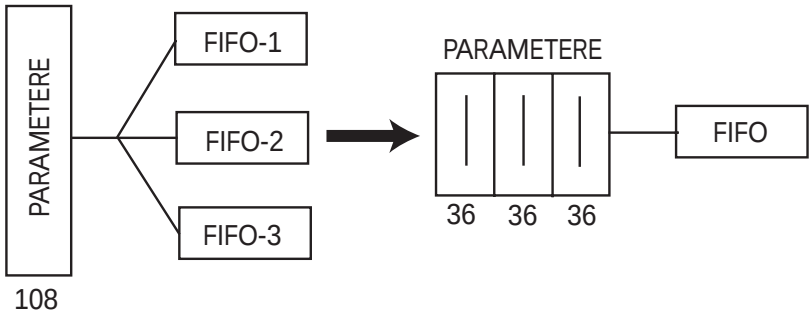
6.2 Ressursbruk

For å redusere plassbruken til adresse-generatoren kan man bytte ut FIFO-køen. Køen som vi bruker har forskjellige skrive og leseklokker, og denne tar mer plass enn en versjon med felles klokke. Den som har felles klokke fikk vi etter mye simulering og testing ikke til å virke. Om det er noe feil med den eller om vi gjorde noe feil, fant vi aldri ut. Derfor valgte jeg å bygge videre på den andre, som virker i simulering. Det går også an å bruke felles klokke på den. Uansett så kan det være kjekt med en FIFO med forskjellig lese og skrive klokke, dersom designet kunne ha kjørt på en klokke hastighet over 200 MHz. Blokkrammen som FIFO-køen bruker, har maks klokkehastighet på 200 MHz, og takler derfor ikke en klokkehastighet på over 200 MHz.

6.3 Parallellitet

For å redusere plassbruken enda mer vil det være naturlig å redusere antall FIFO'er til en, i stedet for de 3 som benyttes. Parametersettet består nå av 6 parametre som opptar 108 IO-pinner. I stedet for å skrive alle parametrene i parallell til 3 FIFO'er, er det hensiktsmessig å skrive dem inn i en FIFO i serie etter hverandre, figur 6.1. Blokkrammen i Virtex-II har den egenskapen at man kan gjøre en "burstwrite" og en "burstread". Det innebærer at *write* signalet som styrer skrivingen går høy i 3 klokkepulser og for hver klokkepuls vil det på inngangen til FIFO'en være en ny del av et parametersett. En skrive og leseoperasjon av parametersettet mot FIFO'køen vil ta 3 klokkecykler i stedet for 1. Dette er overkommelig og vil ikke skape noen problemer for adressegenereringen og bytte av parametersett. Neste parametersett kan klargjøres i bakgrunnen mens addresse-generatoren er opptatt med pågående algoritme. Istedet for at registrene i *reg_contr* er på 108, lager vi 3 registre a 36 bit som mellomlager før settet i parallell blir sendt inn til *addgenerator* parallelt før neste bytte.

Fordelene med de modifiseringer som er nevnt over er at plassbruken blir mindre. En FIFO bruker 69 slices, og to stk. bruker 138. Hele



Figur 6.1: Reduksjon av IOB og plass

designet bruker 592 slices. Å fjerne 2 FIFO køer, vil redusere plassbruken med ca 20 %. Antallet IO-pinner vil bli redusert med ca 2/3 slik at man ikke trenger en stor og dyr krets med mange IO-pinner. På en adresseringskrets vil det også være en del kontrollogikk som også tar en del plassen. Det vil uansett være nok plass til flere adresse-generatorer på en krets, slik at vi oppnår parallellitet ved å generere flere strømmer samtidig slik som beskrevet i kap. 2.1.3. En algoritme kan deles opp i flere mindre deler som eksekveres samtidig ipåfølgende moduler i beregningskretsen. Den totale tiden blir mindre for en gjennomkjøring av algoritmen. Dette kalles for pipelining. Hvert segment av algoritmen vil eksekveres etterhverandre, slik at den totale tiden for algoritmen blir mindre. Hvert datasegment vil bli behandlet (generert) av hver sin adresse-generator. Av relevant arbeid som det er naturlig å sammenligne min løsning med, er følgende:

Adresse-generatoren i [16], SAG, som er implementert på en komersiell FPGA, kan lage flere strømmer samtidig, maks åtte, men vil ikke ha mulighet for å gi så mange datastrømmer ut. Denne begrensningen i antallet strømmer ut, kommer av at RP og DS enhetene på kortet har begrensninger i antall fysiske kanaler. Antallet dimensjoner som kan skannes er 32. Beregningskretsen, RP, på kortet kan ikke alltid jobbe på 66.666 MHz, som er bushastigheten.

En adresse generator er laget i CMOS, i [31] som opererer ved 20 MHz. Denne kan generere adresser uti fra strukturerte data i en, to eller tre dimensjoner. Dersom den er bra programmert med pipelining, kan adresser bli levert hver klokkecykel, men kun i en adressestrøm.

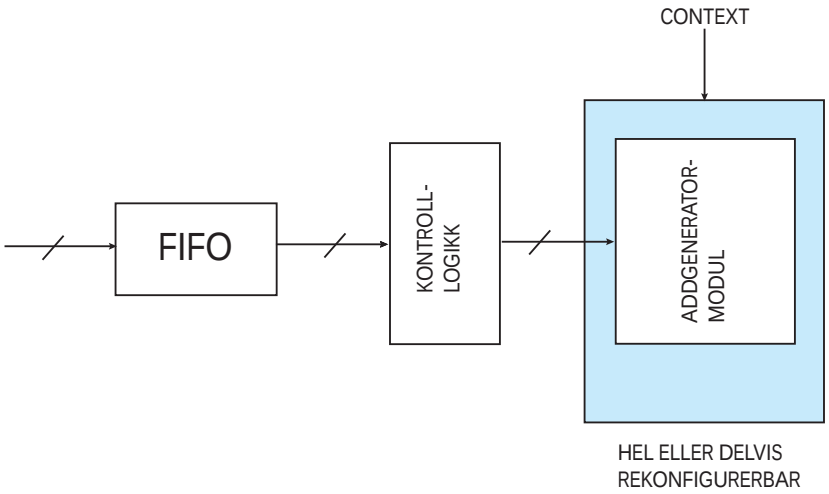
6.4 Rekonfigurerbarhet

Som nevnt i kapittel 4.2.4 kan en avbilding ha mange dimensjoner og være komplisert, med mange subarrayer, step-verdier etc. eller en adressegenerator kan jobbe på data som er organisert i en, to eller tre dimensjonale arrayer. Det finnes derfor mange forskjellige skannemønstre (adressseringsmønstre) slik som 2-dim FIR-filtrering, zig-zag skanning, dataavhengig skanning etc. og det vil være plasskrevende og komplisert å lage en adresse-generator som støtter mange forskjellige avbildninger og skannemønstre. Det eksempelet jeg har konstruert dekker bare et utvalg av interessante adresseringsalgoritmer.

Ved å benytte seg av delvis rekonfigurering av FPGA'en, dvs. "context switching" kan vi bytte forskjellige adreseringsmønster. Dette vil si i praksis at deler av *addgenerator* modulen eller hele er rekonfigurerbar under kjøring, avhengig av adresseringsmønsteret. I første rekke er det tilstandsmaskinen i denne modulen som trenger å reprogrammeres, figur 6.2. Ett adresseringsmønster kan trenge 5 tilstander og 6 parametere, mens et annet trenger 4 tilstander og 4 parametere. Hver context, konfigurasjon, er et skannemønster og kan byttes innimellom addressegenereringen eller midt i. Et context bytte bør ikke ta for lang tid slik at overheaden med et bytte mellom forskjellige skannemønstre blir for stor. Et bytte bør kun ta en klokkecycel, slik at adresse-generatoren ikke trenger å stoppe opp.

Det er laget et forslag på en context switching FPGA i [32] på en kommersiell FPGA, Virtex XCV1000-6. Dette forslaget baserer seg på tidligere arbeider for å implementere en bruker definert FPGA inni en FPGA og innebærer konfigurasjonsbytte under kjøring. Bytte av konfigurasjon vil være mulig på en eller få klokkecykler. En ulempe med forslag som bruker en "virtuell" FPGA for å gjennomføre konfigurasjonsbytte, er at det kreves en del logikk. Mengden logikk som er igjen vil bli mindre, slik som forslagene i [32][35]. Med nye FPGA familier i fra Xilinx, og deres økende mengde med logik, vil det være nok av plass igjen til rekonfigurerbar logikk.

Det er framsatt flere mulige løsninger på å gjennomføre context switching. Noen av dem er er presentert i kap. 3.6. Det har ikke vært mulig å se på hvilken løsning til "context switching" som er best å bruke og implementering av konfigurasjonsbytte pga. tidsrammen til dette prosjektet.



Figur 6.2: Addresse-generator med context switching

6.5 Videre arbeid

- **Utvidelser og videre arbeid**

  -Implementere hele designet og teste det på labkortet. Testen bør ta for seg både funksjonalitet og hastighet.

  -Utvide for parallellitet og pipelining.

  -Se på mulige løsninger for en adresseringskets på en Virtex FPGA med context switching og implementere og teste en slik løsning på den fysiske kretsen.

Kapittel 7

Konklusjon

Det finnes forskjellige løsninger og plattformer for å implementere beregninger med høye krav til ytelse. Vi har sett på noen løsninger som er laget for slike operasjoner, det være seg på en ASIC, CPU, kommersielle FPGA'er eller egenproduserte plattformer. Ofte er dataene strukturert lagret i minnet slik at man kan benytte seg av dataflytmodellen i stedet for den vanlige von Neumann modellen. For å lage et design med en dataflytmodell trengs det en adresse-generator.

Vi har i denne oppgaven konstruert og simulert en adresse-generator som takler en avbildning i 3-dim på en kommersiell FPGA som genererer en strøm av data til en beregningskrets for hver klokkepuls, såkalt strømbasert beregning. Konstruksjonen er beskrevet i VHDL. Det er utført simulering, syntese og "place and route". I tillegg så er en del av designet testet på et utviklingskort. Implementasjonen viser at man kan oppnå en estimert klokkehastighet på 144 på en Virtex-II pro, som videre vil gi høy ytelse.

Etter en nærmere analyse av implementasjonen og utviklingen i FPGA-teknologien de siste årene, har implementasjonen et potensiale til videreutvikling som vil gi en høyere hastighet og større fleksibilitet. Videre arbeid vil være å utnytte de fordelene parallellitet og pipelining har og implementere dette. Å inkorporere "context switching" i designet vil gi en høy grad av fleksibilitet, der adresse-generatoren kan lage adresser ut i fra et veldig stort antall avbildninger og skannemønstre.

Bibliografi

[1] *M. Gokhale et al. Splash. A reconfigurable Linear Logic Array; International Conference on Parallel Processing pp. 526-532, 1990*

[2] *J. Vuillemin et al. Programmable Active Memories. Reconfigurable Systems Come of Age; IEEE Transactions on VLSI Systems, Vol 4, No. 1, pp. 9-16, 1993*

[3] *R. W. Hartenstein. A New Business Model - and its Impact on Soc Design; Euromicro Symposium on Digital Systems Design (DSD'01), September 04–06, 2001, Warsaw, Poland*

[4] *Egenskaper til Virtex-IIPro; http://www.xilinx.com/xlnx/xil_prodcat_landingpage.jsp?title=Virtex-II+Pro\FPGAs*

[5] *Rolf Fiedler. Beyond ILP- Trends for Programmable DSP Machines; 17 april 2001*

[6] *R. W. Hartenstein, A. G. Hirschbiel, M. Weber. MoM-Map Oriented Machine; in: Ambler et al. : (Prepr.Int'l Worksh. on) Hardware Accelerators, Oxford 1987, Adam Hilger, Bristol 1988*

[7] *R. W. Hartenstein, A. G. Hirschbiel, M. Riedmuller, K, Schmidt, M. Weber. A High Performance Machine Paradigm Based on Auto-Sequencing Data Memory; 1991*

[8] *R. W. Hartenstein, J. Becker, M. Herz, U. Nageldinger. A General Approach in System Design Integrating Reconfigurable Accelerators; Proc.IEEE Int'l Conf. on Innovative Systems in Silicon; Austin, TX, Oct. 1996*

[9] *R. W. Hartenstein, J. Becker, M. Herz, U. Nageldinger. An Embedded Accelerator for Real World Computing; in Proceedings of IFIP International Conference on Very Large Scale Integration, VLSI'97, Gramado, Brazil, August26-29, 1997*

[10] *J. Villasenor, W. H. Mangione-Smith. Configurable computing; Scientific American, no. 6, 1997*

[11] *J. Torresen. Reconfigurable Logic Applied for Designing Adaptive Hardware Systems; International Conference on Advances in Infrastructure for Electronic Business, Education, Science, and Medicine on the Internet (SSGRR 2002W), January 2002, L'Aquila, Italy*

[12] *R. W. Hartenstein. A decade of reconfigurable computing: a visionary retrospective; in Proceedings of international Conference on Design Automation and Testing in Europe and Exhibit(DATE), 2000, Munchen*

[13] *T. Fuji et al. A dynamically reconfigurable logic engine with a multi-context multi-mode unified cell architecture ; in Proceedings of Int. Solid-State Circuits Conf, 1999, pp. 360-361*

[14] *R. Sidhu et al. A self-reconfigurable gate array architecture; in Field-programmable Logic and Applications 10th int. Conference on Field-Programmable Logic and Applications(FPL-2000), R.W.Hartenstein et al., Eds., Lecture Notes in Computer Science, vol.1896, pp.106-120, Springer-Verlag, 2000*

[15] *Scott Hauck. The Roles of FPGAs in Reprogrammable Systems; Proceedings of the IEEE, Vol. 86, No.4, pp.615-639, 1998*

[16] *V. Michael Bove et al. Media Processing with Field-Programmable Gate Arrays on a Microprocessor's local Bus; Proc. SPIE Media Processors, 3655, 1999*

[17] *G. Kahn. The semantics of a Simple Language for Parallel Programming; Proceedings of the IFIP Congress, Aug. 1979, pp. 35-45*

[18] *W. H. Burge. Stream Processing Functions; IBM Journal of Research and Development, 19 Jan. 1975, pp. 12-25*

[19] *J. A. Watlington and V. M. Bove, Jr. Stream-based Computing and Future Television; SMPTE Journal, 106, April 1997, pp. 217-224*

[20] *Kevin Skahill. VHDL for Programmable Logic; Addison-Wesley Publishing Company, Inc, pp. 25-53*

[21] *David Van den Bout. The Practical Xilinx Designer Lab Book; Prentice-Hall International Limited, pp. 12-30*

[22] *Virtex-II Pro Detailed Functional Description (Module 2) http://www.xilinx.com/xlnx/xweb/xil_publications_display.jsp?BV_SessionID=@@@@1222611363.1078146086@@@@&BV_*

EngineI\%D=ccceadckkkhikdicflgcefldfgldgji.0&sGlobal NavPick=&sSecondaryNavPick=&category=-18773&iLanguageID=1

[23] *Frequently-Asked Questions (FAQ) About Programmable Logic; http://www.optimagic.com/faq.html*

[24] *R. W. Hartenstein et al. A novel paradigm of parallel computation and its use to implement simple high-performance hardware; Future Generation Computer Systems 7 (1991/92) 181-198, North-holland*

[25] *Generelle spesifikasjoner til Intel Pentium serien http://www.globalspec.com/help/spechelp.html?name=Handheld_ Computers&comp=4075&sectionid=2*

[26] *Carl Ebeling et al. RaPid-A Configurable Computing Architecture for Compute-Intensive Applications; University of Washington. Technical Report UW-CSE-96-11-03, Nov. 1996*

[27] *Seth Copen Goldstein et al. PipeRench: A Coprocessor for Streaming Multimedia Acceleration; Proc. of The 26th Annual International Symposium on Computer Architecture, May 1999, Atlanta, Georgia*

[28] *Oversikt over spesifikasjoner til noen Xilinx-kretser; http://www.xilinx.com/products/tables/fpga.htm#v2p*

[29] *Geir Nilsen. A Variable Word-Width Content Adressable Memory(CAM) for Fast String Matching; Cand. Scient. Report*

[30] *John A. Watlington and V. Michael Bove, Jr. A System for Prallel Media Processing ; Parallel Computing, 23:12 Dec. 1997, pp. 1793-1809*

[31] *R. W. Hartenstein and Helmut Reinig. Novel Sequencer Hardware for High-Speed Signal Processing Workshop on Design Methodologies for Microelectronics, Smolenice Castle, Slovakia, September 1995*

[32] *Jim Torresen and Knut Arne Vinger; High Performnace Computing by context switching Reconfigurable Logic In proc. of 16th European Simulation Multiconference (ESM-2002), pp. 207–210, June 2002, Darmstadt, Germany*

[33] *S. Scalera. and J. Vazques. The Design and Implementation of a Context Switching FPGA In IEEE Symposium on FPGAs for Custom Computing Machines, pp 78-85, 1998*

[34] *Memec Design; http://www.memecdesign.com*

[35] *L. Sekanini and R.Ruizicka. Design of the Special Fast Reconfigurable Chip Using Common FPGA In Proc. of Design and Diagnostics of Electronic Circuits and Systems IEEE DDECS'2000,pp 161-168*

[36] *Forbindelses Spesifikasjoner til Virtex-II pro http://www.xilinx.com/xlnx/xil_prodcat_product.jsp?title=v2p_systemio*